ViewPoint Programmer's Manual

XEROX

610E00191
October 1988

Xerox Corporation
XDE Technical Services, MS SV403
475 Oakmead Parkway
Sunnyvale, California 94086

Table of Contents

50 StarWindowShell

I.

SYSTEM LEVEL INTERFACES

1

Introduction

This *ViewPoint Programmer's Manual* is written for programmers who are developing applications to run on ViewPoint software. ViewPoint's open architecture philosophy allows applications to be developed easily.

You will find this manual useful only if you are already a Mesa programmer. You should have completed the Mesa Course and be familiar with the contents of the *XDE User's Guide* (610E00140) and the *Mesa Language Manual* (610E00170). You should also be familiar with the facilities described in the *Pilot Programmer's Manual* (610E00160) and the *Filing Programmer's Manual* contained in the *Services Programmer's Guide* (610E00180).

The *ViewPoint Programmer's Manual* gives you the information you will need to implement the user interface of an application that runs on ViewPoint. This includes how to:

- Represent applications as icons.
- Interact with the mouse and keyboard to process the user's instructions.
- Create folder-like containers.
- Create property sheets.
- Create menus.
- Paint pictures and text on the display.
- Create programmable keyboards.
- Represent and manipulate multinational text.

It does not provide you with Mesa, Pilot, or Services-specific information.

1.1 Document Structure

This introductory chapter describes the physical manual itself, how it is organized, who should read it, how it should be read, and why. Chapter 2, Overview, describes ViewPoint and discuss its history and overall design.

Chapter 3, The Programmer's Guide, tells how to use the ViewPoint interfaces. It describes concepts essential to understanding ViewPoint and describes the facilities that are available. The most common interfaces are briefly discussed and grouped by application. All of the ViewPoint interfaces, with a short summary, are listed alphabetically at the end of the chapter.

The individual interface chapters are arranged alphabetically in Chapters 4 through 59. These chapters give detailed descriptions of the interfaces that ViewPoint provides. Each interface chapter begins with an overview that explains the concepts behind the interface and the important data types that it manipulates. The second section of each chapter describes the actual items of the Mesa interface and groups them by function. The third section explains typical ways of using the interface and often contains programming examples. The fourth section is the index of interface items. Within an interface chapter, the items of the broadest interest are presented first; more specialized items follow later.

Appendix A presents the system TIP Tables, references are in Appendix B, Appendix C contains a list of well-known atoms, and Appendix D contains a listing of public symbols.

1.2 Getting Started

Chapters 1, 2, and 3 of the *ViewPoint Programmer's Manual* should be read in order. Within Chapter 3, you will sometimes be guided to various sections in task-relative rather than page-relative order. Chapters 4 through 59 (the interface chapters) can be read in any order, depending on your need.

2 Overview

2.1 What Is ViewPoint?

ViewPoint is a collection of facilities for writing application programs that run on a personal workstation with a high-resolution bitmap display. It supports an open–ended collection of applications, providing a framework and a set of rules that allow these independent applications to be integrated. It has an advanced user interface that also allows applications to be easily adapted for users in other countries.

Throughout this document, the term *user* describes a person who interacts with the applications built on ViewPoint via the mouse and keyboard. Programs cannot predict or control user actions. The term *client* describes programs that use the facilities described in this document. The client may act as a result of some user action, but the behavior of the client is the result of a program and under control of its implementor.

2.1.1 User Abstractions

ViewPoint uses several abstractions that are part of the advanced user interface pioneered by the Star Workstation:

- *Icons and Desktop*. Icons that represent objects on a desktop are one basic abstraction. These objects can represent either functions or data. Data icons, such as a document, represent objects on which actions can be performed. Function icons, such as a printer, represent objects that perform actions. In the metaphor, they are on the desktop that also serves as the background for their display. With ViewPoint, clients may create new icons that provide additional functions within the desktop metaphor.

- *Windows*. Windows are rectangular areas on the screen that display the contents of an icon when it is opened. Each window has a header containing the name of the window's icon and a set of commands. The window also contains scroll bars that scroll the contents of the window vertically and horizontally.

- *Property Sheets*. Property sheets are displayed forms that show the properties of an object. They contain several types of parameters, including state parameters, which may be on or off; choice parameters, which have a set of mutually exclusive values; and text parameters.

- *Selection.* The selection is an object or body of data identified by the user. It is the target of user actions; there can be only one selection at any one time. It can be a string of text that the user may then delete, copy, or change the properties of. It can be an icon on the desktop that is moved to a printer icon for printing or opened to display its contents. In general, it can be almost any piece of data that can be represented on the screen.

2.1.2 Client Abstractions

To implement the above user abstractions and to provide some building blocks for developing applications, ViewPoint uses several client abstractions:

- *Containee and StarDesktop.* **Containee** is an application registration facility that associates an application with a file type. Registering an application consists of providing procedures that paint iconic pictures and perform various operations. **StarDesktop**, using the desktop metaphor, displays the desktop window and iconic pictures for each file found in a particular directory.

- *Client Windows.* The client window abstraction is more primitive than the user window abstraction. The client window abstraction serves to isolate applications from the physical display and each other. A window can be thought of as a quarter of an infinite plane. Within that space, the client is called upon to display the contents of the window without regard to any other applications' windows. Windows may be linked to form a tree structure. A user's window is typically composed of a number of small client windows–one for the header, one for each scroll bar, and so forth.

- *Menus.* Menus are sequences of named commands, each consisting of a text name and a procedure. Menus may be displayed to the user in several forms, such as in a pop–up menu or as window shell header commands (see below).

- *Window Shells.* The user window abstraction is implemented by window shells. They provide the header, scroll bars, and body windows. The body windows are windows the client uses to display the content of an application. The commands in the header are menus.

- *Form Windows.* Form windows are the client abstraction that provides the basis for the user property sheet. Form windows allow form items in a window to be created and manipulated. There are several types of items: boolean items, choice items, text items, numeric text items, command items, form and window items. Window items allow the client to implement its own type of item. The property sheet user abstraction is implemented by putting a form window inside a window shell.

- *Container Windows.* Container windows implement a window that contains a list of items. Clients supply the source of items and the container window handles that display the contents in a window and interact with the user.

- *Selection.* The client selection abstraction is a framework in which a client can manifest itself as the holder of the user's current selection while other clients interrogate the selection and request that it be converted to a variety of data types. ViewPoint defines several selection conversion types, but the selection framework allows clients to define additional conversion types. The selection is the principal means by which information is transferred between different applications.

2.1.3 System Structure

ViewPoint's architecture contains a small set of public interfaces that provide the basic facilities for building workstation applications. Facilities are included in ViewPoint for several reasons. Some facilities implement system-wide features, such as the window package. If several applications tried to implement their own window packages, chaos would result. Facilities are also included in ViewPoint to provide a consistent user interface, such as form windows and property sheets. A final reason for including facilities is to provide packages that are useful to many clients, such as the simple text facilities. As ViewPoint evolves, more facilities useful to a variety of clients will be added.

The ViewPoint interfaces fall into the following general categories:

Application registration:	**Containee**
Windows and display:	**Context, Display, StarWindowShell, Window**
Forms and property sheets:	**FormWindow, FormWindowMessageParse, PropertySheet**
User input and keyboards:	**BlackKeys, KeyboardKey, KeyboardWindow, LevelIVKeys, SoftKeys, TIP, TIPStar**
Strings and messages:	**XChar, XCharSets, XCharSetNNN, XComSoftMessage, XFormat, XLReal, XMessage, XString, XTime, XToken**
Selection:	**Selection**
Containers:	**ContainerCache, ContainerSource, ContainerWindow, FileContainerShell, FileContainerSource**
Text display and editing:	**SimpleTextDisplay, SimpleTextEdit, SimpleTextFont**
Background management:	**BackgroundProcess**
Miscellaneous user interface:	**Attention, Cursor, MenuData, MessageWindow, PopupMenu, StarDesktop, Undo**
Miscellaneous:	**Atom, AtomicProfile, Event, IdleControl**

2.2 History

ViewPoint is the result of past experience with Star and the Xerox Development Environment. In late 1982, the Star Performance and Architecture Project concluded that Star's monolithic system structure, in which every piece knew about every other piece, hindered its performance. The monolithic structure also made it difficult to develop new applications. In addition, there were hundreds of interfaces in the system but no distinction between public and private interfaces, which made it difficult for programmers to learn how to write applications in the system.

In contrast to Star, the Xerox Development Environment had a modular system structure with a small number of well-documented public interfaces It also encouraged an open-ended collection of applications. While it performed well and was open, the Xerox Development Environment did not have as consistent a user interface as Star, nor did it support Star's multilingual requirements.

As a result of this study, ViewPoint was created. It has the system structure, documented public interfaces, and openness of the Xerox Development Environment, yet supports Star's user interface and multilingual requirements.

While it was initially focused on providing a new foundation for Star, ViewPoint has become the basis for more software products from the Office Systems Division. It will evolve to replace the current foundation of the Xerox Development Environment and will likely support products from organizations outside the Office Systems Division.

2.3 Philosophy and Conventions

ViewPoint's philosophy and conventions apply both to applications that interact with the user and to packages that implement a facility. Some are just good system-building concepts. ViewPoint assumes that programs that run within it are friendly and that they are not trying to circumvent or sabotage the system. The system does not try to enforce many of these conventions but assumes that clients will adhere to them voluntarily. If these conventions are not followed, the system may degrade or break down altogether.

2.3.1 Supported Public Interfaces

Systems should be designed to export public interfaces that are well documented and relatively stable. By defining a set of primitive facilities and stressing their stability, applications are encouraged to depend on the existing ViewPoint facilities rather than on other applications packages. This promotes an *open architecture* in which applications can be developed and loaded with relative ease, exchanging information among themselves while maintaining the independence of client modules. The open architecture allows designing for unknown applications as well as the class of applications expected in Star.

In keeping with an open architecture, ViewPoint does not make far-reaching assumptions about the applications that run above it. While ViewPoint provides facilities that make certain styles of applications easy, it does not preclude other styles of applications.

2.3.2 Plug-ins

ViewPoint is self-contained in that it does not import procedures that it expects a client to supply. Rather it waits, in effect, for clients to call it and state that they want to implement some facility. This is referred to as a *plug-in approach*: an application plugs itself in to a lower layer of software.

Plug-ins encourage modularity at the client level. Because ViewPoint can be run by itself (although it does not do much), it can also be run with just one application plugged in. Thus each application can be implemented and debugged individually, which simplifies system development.

Plug-ins also can break a dependency that would create a complex dependency graph. For example, the desktop has a dependency on the applications that appear in the desktop. If the desktop depended directly on the applications, it would have to change every time a new application was created. By having the applications plug themselves into the desktop, the direct dependency is broken.

2.3.3 Don't Preempt the User

Clients should avoid dictating what the user must do. The user should be free to interact with different applications as desired. For example, the current selection is something that the user should control. It should be changed only as a result of user actions. A background process should not change the selection out from under the user.

2.3.4 Don't Call Us, We'll Call You

Because the user is in control, a program must wait for the user to interact with it. The method of interacting with the user that is prevalent in terminal-oriented user interfaces is to get a command from the user and execute it, which results in the client regaining control while it awaits user input. With potentially multiple applications active simultaneously, the user should be free to interact with the one of his choosing. ViewPoint's input facilities notify a window when the user inputs to that window.

Events are another case in which the system calls the client. For example, a client may need to do something when the user logs in. If the client registers a procedure with the appropriate event, the procedure is invoked when the event occurs.

3

Programmer's Guide

This ViewPoint application programmer's guide is intended to point the programmer to the most important parts of the most important interfaces needed for writing an application in ViewPoint.

ViewPoint is a collection of interfaces to be used for writing application programs. It is primarily intended to support applications like those in the ViewPoint workstation; that is, there is support for icons, windows, property sheets, and so forth.

The first section (3.1 Guide) contains a jump table of the form, "If your application does X, then you use interfaces A and B; also, you need to understand C and D, and you probably want to read section 3.1.x." The subsections (3.1.x) provide more detail about A, B, C, and D, pointing the programmer to the most important types and procedures in an interface. The second section (3.2 Getting Started) contains essential information for first-time ViewPoint programmers. Section 3.3 provides some flow of control descriptions for several common scenarios. It describes which interfaces call which client procedures when, and so forth. Section 3.4 discuss some programming conventions specific to ViewPoint interfaces. Section 3.5 contains a summary of all the ViewPoint interfaces.

First, we briefly define an application from the user's point of view: The user sees the icons on the desktop and can operate on them in various ways. You can select an icon with the mouse and open it to display its contents. Or by selecting the icon and pressing PROPS, you can examine and change the icon's properties through a window called a *property sheet*. After an icon is opened, he can examine the properties of the contents and change them by again using the property sheet. By selecting one icon, pressing COPY or MOVE, and then selecting another icon, he can perform various application-specific operations. This is often referred to as "dropping one icon onto another." Each application attaches a different meaning to the drop-on operation. For example, the folder takes the icon dropped onto it and adds it to the folder. The printing application (printer icon) prints the icon dropped onto it.

From the application's point of view, an icon is just a picture that represents a file. Files have a file type, and an application operates on all files of the same type. Thus when the user selects a folder icon, he or she is actually selecting a file with file type of folder. When the user performs some operation on an icon, the desktop calls the appropriate application based on the file type of the file the selected icon represents.

3.1 Guide

The following table can help you readily find a desired section.

3.1.1 Guide to the Guide

If your application ...	**See section**
... Appears as an icon:	
- Read about icon applications in **3.2 Getting Started**	3.2.2
- Use **Containee** to register the icon's behavior	3.1.2
... Opens a window:	
- Use **StarWindowShell** to create a window	3.1.3
- Use **MenuData** to construct menus	3.1.4
... Manages the contents of a window:	
- Use **Display** and **Window** to display information	3.1.5
- Supply a **TIP.NotifyProc** to process user actions	3.1.5
- Use **Selection** to share data between applications	3.1.5
- Use **Context** to save data with the window	3.1.5
... Puts up a Property Sheet:	
- Use **PropertySheet** and **FormWindow** interfaces	3.1.6
... Manipulates strings:	
- Use the **XString** interfaces (including **XFormat**, **XToken**, **XChar**)	3.1.7
... Displays messages to the user:	
- Use the **XMessage** and **Attention** interfaces	3.1.8
... Displays a list of items like a folder:	
- Use the Container interfaces (**ContainerWindow**, **ContainerSource**)	3.1.9
... Redefines the function keys:	
- Use the **SoftKeys** interface	3.1.10
... Redefines the Black Keys:	
- Use **BlackKeys** and **KeyBoardKey** interfaces	3.1.11
... Performs operations in a background process:	
- Use the **BackgroundProcess** interface	3.1.12

3.1.2 Containee

Containee is an application registration facility. An *application* is a software package that implements the manipulation of one type of file. **Containee** is a facility for associating an application with a file type. (§3.2.2 explains how an application registers itself and is then invoked to perform various operations). The most important items in **Containee** are:

Implementation	A record containing several client procedures.
SetImplementation	Registers an application.
GenericProc	Client procedure called to perform OPEN, PROPS, COPY/MOVE-onto, and so forth.
PictureProc	Client procedure called to display an icon picture.
Data, DataHandle	Uniquely identifies a file.

3.1.3 Application Windows

StarWindowShell allows a client to create a Star-like window. A **StarWindowShell** window has a header that contains a title, commands, and pop-up menus. The window may have scroll bars, both horizontal and vertical. It also has interior window space that may contain anything the client desires. **StarWindowShell** also supports the notion of opening within.

A **StarWindowShell** is a window (see **Window** interface) that is a child of the desktop window. A **StarWindowShell** has an interior window that is a child of the **StarWindowShell** and is exactly the size of the available window space in the shell, that is, the window shell minus its borders and header and scrollbars. The interior window may have child windows created by the client. These children of the interior window are called *body windows*. The client may create an arbitrary number of body windows and may arrange them arbitrarily. **Note:** Because the body windows are children of the interior window, they are clipped by the interior window.

The client may manage body windows directly, including all display and notification (user input). Body windows can also be managed by various interfaces provided by ViewPoint, such as **FormWindow** and **ContainerWindow**. These interfaces have **Create** procedures that take a body window and turn it into a particular kind of window, providing all the display and notification handling for the window.

The most important items in **StarWindowShell** are:

Create	Creates a **StarWindowShell** window.
CreateBody	Creates a body window.
ShellFromChild	Returns the window shell, given a body window.
SetRegularCommands	Places commands in the header of a **StarWindowShell**.

AddPopupMenu	Adds a pop-up menu to the header of a **StarWindowShell**.

3.1.4 Menus

A *menu* is a list of named commands. When the user selects a menu command, a client procedure is called. The **MenuData** interface allows menu items and menus to be created. **MenuData** does not address the user interface for menus. Menu items may appear as commands in the header of a star window shell (**StarWindowShell.SetRegularCommands**). Entire menus may be accessed via a pop-up symbol in the header of a window shell (**StarWindowShell.AddPopupMenu**). Menu items may be added to the pop-up menu that is available to the user through the attention window (**Attention.AddMenuItem**).

The most important items in **MenuData** are:

CreateItem	Creates a menu item.
MenuProc	A client procedure that is called when the user selects a menu item.
CreateMenu	Creates a menu from an array of menu items.

3.1.5 Managing a Body Window

Clients can manage their own body windows. This involves handling both display and notification (user-input), and often includes managing the current selection. Display is done by providing a window display procedure. Notifications are received through a client-provided **TIP.NotifyProc**. The **Selection** interface manages the current selection. Arbitrary data associated with a window can be saved with the window by using the **Context** interface.

3.1.5.1 Display

The **Window** interface calls the client's display procedure to repaint the contents of the window. It is called when the window is initially made visible. It is also called when the window suddenly becomes more visible because an overlapping window was moved, or when the window is scrolled so that the part of it that was invisible before becomes visible. The display procedure should use the **Display** and/or **SimpleTextDisplay** interfaces to display bits in the window. The display procedure can be set when a window shell's body window is created (**StarWindowShell.CreateBody**) or by calling **Window.SetDisplayProc**.

The most important item in **Window** is the client's display procedure. There is no TYPE for this procedure, but it is discussed in the **Window** interface chapter. Other important items:

Box	Defines a rectangle in a window.
Place	Defines a point in a window.

The most important items in **Display** are:

Black	Displays a black box.
White	Displays a white box.
Invert	Inverts the bits in a box.
Bitmap	Displays an arbitrary array of bits.

The most important item in **SimpleTextDisplay** is:

StringIntoWindow	Displays a string in a window.

3.1.5.2 TIP and TIPStar

TIP provides basic user input facilities through a flexible mechanism that translates hardware-level actions from the keyboard and mouse into higher-level client action requests (result lists). The acronym TIP stands for *terminal interface package*. This interface also provides the client with routines that manage the input focus, the periodic notifier, and the **STOP** key.

The basic notification mechanism directs user input to one of many windows in the window tree. Each window has a **TIP.Table** and a **TIP.NotifyProc**. The table is a structure that translates a sequence of user actions into a sequence of results that are then passed to the notify procedure of the window.

The Notifier process dequeues user events, determines which window the event is for, and tries to match the events in the window's **Table**. If it finds a match in the table, it calls the window's **NotifyProc** with the results specified in the table. If no match is found, it tries the next table in the window's chain of tables. If no match is found in any table, the event is discarded.

TIP tables provide a flexible method for translating user actions into higher-level client-defined actions. They are essentially large select statements with user actions on the left side and a corresponding set of results on the right side. Results may include mouse coordinates, atoms, and strings for keyboard character input.

ViewPoint provides a list of normal tables that contain one production for each single user action. Client programmers can write their own table to handle special user actions and link it to system-defined tables, letting those tables handle the normal user actions. These system-defined tables are accessible through the **TIPStar** interface and are described in **Appendix A**.

Input Focus. The input focus is a distinguished window that is the destination of most user actions. User actions may be directed either to the window with the cursor or to the input focus. Actions such as mouse buttons are typically sent to the window with the cursor. Most other actions, such as keystrokes, are sent to the current input focus. Clients may make a window be the current input focus and be notified when some other window becomes the current input focus.

The current selection and the current input focus often go together. If the window in which a selection is made also expects to receive user keystrokes (function keys as well as black keys), **TIP.SetInputFocus** should be called at the same time as **Selection.Set** is called. This is also the time to call **SoftKeys.Push** or **KeyboardKey.RegisterClientKeyboards**, if necessary.

Modes. **TIPStar** also provides the notion of a global mode to support MOVE, COPY, and SAME. When the user presses down and releases the MOVE, COPY, or SAME keys, the client that currently has the input focus will receive the notification and should call **TIPStar.SetMode**. This changes the mouse TIP table so that atoms specific to the mode are produced rather than normal atoms when the user performs mouse actions. For example, in copy mode "CopyModeDown" instead of "PointDown" is produced when the user presses the left mouse button. This informs the client that receives the atom that it should attempt to copy the current selection rather than simply select something.

The most important items in **TIP** are:

NotifyProc	Client procedure that is called to handle a user action.
Results, ResultObject	Right side of the table entry that matched the user action.
SetInputFocus	Sets a window to be the current input focus.

The most important items in **TIPStar** are:

NormalTable	Returns the chain of system-provided **TIP** tables.
SetMode	Sets the entire environment into MOVE, COPY, or SAMEAS mode, thus changing the results produced for mouse clicks.

3.1.5.3 Context

The **Context** interface allows arbitrary client data to be associated with a window. Client data is usually allocated and associated with the window when the window is created. The data may be retrieved any time, such as at the beginning of the client's display procedure and **TIP.NotifyProc**.

The most important items in **Context** are:

Create	Associates data with a window.
Find	Recovers the data previously associated with a window.

3.1.5.4 Selection

The **Selection** interface defines the abstraction that is the user's current selection. It provides a procedural interface to the abstraction that allows it to be set, saved, cleared, and so forth. It also provides procedures that enable someone other than the originator of the selection to request information relating to the selection and to negotiate for a copy of the selection in a particular format.

The Selection interface is used by two different classes of clients. Most clients wish merely to obtain the value of the current selection in some particular format; such clients are called *requestors*. These programs call Convert (or maybe ConvertNumber, which in turn calls Convert), or Query, or Enumerate. These clients need not be concerned with many of the details of the Selection interface.

The other class of clients are those that own or set the current selection; these clients are called *managers*. A manager calls Selection.Set and provides procedures that may be called to convert the selection or to perform various actions on it. The manager remains in control of the current selection until some other program calls Selection.Set. These clients need to understand most of the details of the Selection interface.

A client that is managing its own body window will be both a selection requestor and a selection manager in different parts of the code. For example, when the user selects something in another window and copies it to the client's window, the client must call Selection.Convert to request the value of the selection in a form appropriate to the application. On the other hand, when the user clicks a mouse button in the client's window, the client usually becomes the selection manager by calling Selection.Set.

The most important items in Selection are:

Convert	Request the value of the selection in some target form.
Value	A record containing a pointer to the converted selection value, among other things.
CanYouConvert	Returns TRUE if the selection manager can convert the selection to a particular target type.
Set	Called by a selection manager to become the current manager.
ConvertProc	Manager-supplied procedure that will be called to convert the selection to some target type.
ActOnProc	Manager-supplied procedure that will be called to perform some action on the selection, such as mark, unmark, clear.

3.1.6 Property Sheets and FormWindow

A property sheet shows the user the properties of an object and allows the user to change these properties. There are several different types of properties, the most common ones being boolean, choice (enumerated), and text.

From a client's point of view, a property sheet is simply a **StarWindowShell** with a **FormWindow** as a body window. A property sheet is created by calling **PropertySheet.Create**, providing a procedure that will make the form items in the **FormWindow** (a **FormWindow.MakeItemsProc**), a list of commands to put in the header of the property sheet, such as Done, Cancel, and Apply (**PropertySheet.MenuItems**), and a procedure to call when the user selects one of these commands (a **PropertySheet.MenuItemProc**). When the user selects one of the commands in the header of the property sheet, the client's **PropertySheet.MenuItemProc** is called. If the user selected Done, for example, the client can then verify and apply any changes the user made to the object's properties.

The most important items in **PropertySheet** are:

Create	Creates a property sheet.
MenuItems	Used for specifying which commands to put in the header of the property sheet.
MenuItemProc	Client procedure called when the user selects one of the commands in the header.

The most important items in **FormWindow** are:

MakeItemsProc	Client procedure called to create the items in the form.
MakeXXXItem	Makes a form item. **XXX** can be **Boolean**, **Choice**, **Text**, **Integer**, **Decimal**, **Window**, **TagOnly**, **Command**.
GetXXXItemValue	Returns the current value of an item. **XXX** can be **Boolean**, **Choice**, **Text**, **Integer**, **Decimal**, **Window**, **TagOnly**, **Command**.

3.1.7 XString, et al.

The *Xerox Character Code Standard* defines a large number of characters, encompassing not only familiar ASCII characters but also Japanese and Chinese Kanji characters and others to provide a comprehensive character set able to handle international information processing requirements. Because of the large number of characters, the data structures in **XString** are more complicated than a LONG STRING's simple array of ASCII characters, but the operations provided are more comprehensive.

Characters are 16-bit quantities that are composed of two 8-bit quantities, their character set and character code within a character set. The Character Standard defines how characters may be encoded, either as runs of 8-bit character codes of the character set or as 16-bit characters where the character set and character code are in consecutive bytes. (See the **XChar** chapter for information and operations on characters.)

ViewPoint provides a string package consisting of several interfaces that support the *Xerox Character Code Standard*. **XString** provides the basic data structures for representing encoded sequences of characters and some operations on these data structures. **XFormat** converts other TYPEs into **XStrings**. **XToken** parses **XStrings** into other TYPEs. **XChar** defines the basic character type and some operations on it. **XCharSets** enumerates the character sets defined in the Standard. A collection of interfaces enumerate the character codes of several common character sets (**XCharSetNNN**). **XTime** provides procedures to acquire and edit times into **XStrings** and **XStrings** into times.

3.1.8 XMessage and Attention

XMessage supports translation into other languages of text displayed to the user. It does not include any string constants in the code of an application. Rather, all the string constants for an application are declared in a separate module and registered with **XMessage**. Then whenever the application needs a string constant, it obtains it by calling **XMessage.Get**. Several commonly used messages such as "Yes", "No", and days of the week are defined in **XComSoftMessage**.

The most important items in **XMessage** are:

Get	Retrieves a message.
RegisterMessages	Registers all the messages for an application.

The **Attention** interface provides a global mechanism for displaying messages to the user. **Attention** provides procedures to post messages to the user in the attention window, clear the attention window, post a message and wait for confirmation, and so forth.

The most important items in **Attention** are:

Post	Posts a message in the attention window.
Clear	Clears the attention window.
formatHandle	**XFormat.Handle** that may be used to format strings into the attention window.

3.1.9 Containers

The Container interfaces (**ContainerSource**, **ContainerWindow**, **FileContainerSource**, **FileContainerShell**, and **ContainerCache**) provide the services needed to implement an application that appears as an ordered list of items to be manipulated by the user. Star Folders are a typical example of such an application.

Figure 3-1 shows the relationships among the various interfaces and potential clients. Each interface is described below, followed by a discussion of which interfaces an application might need to use.

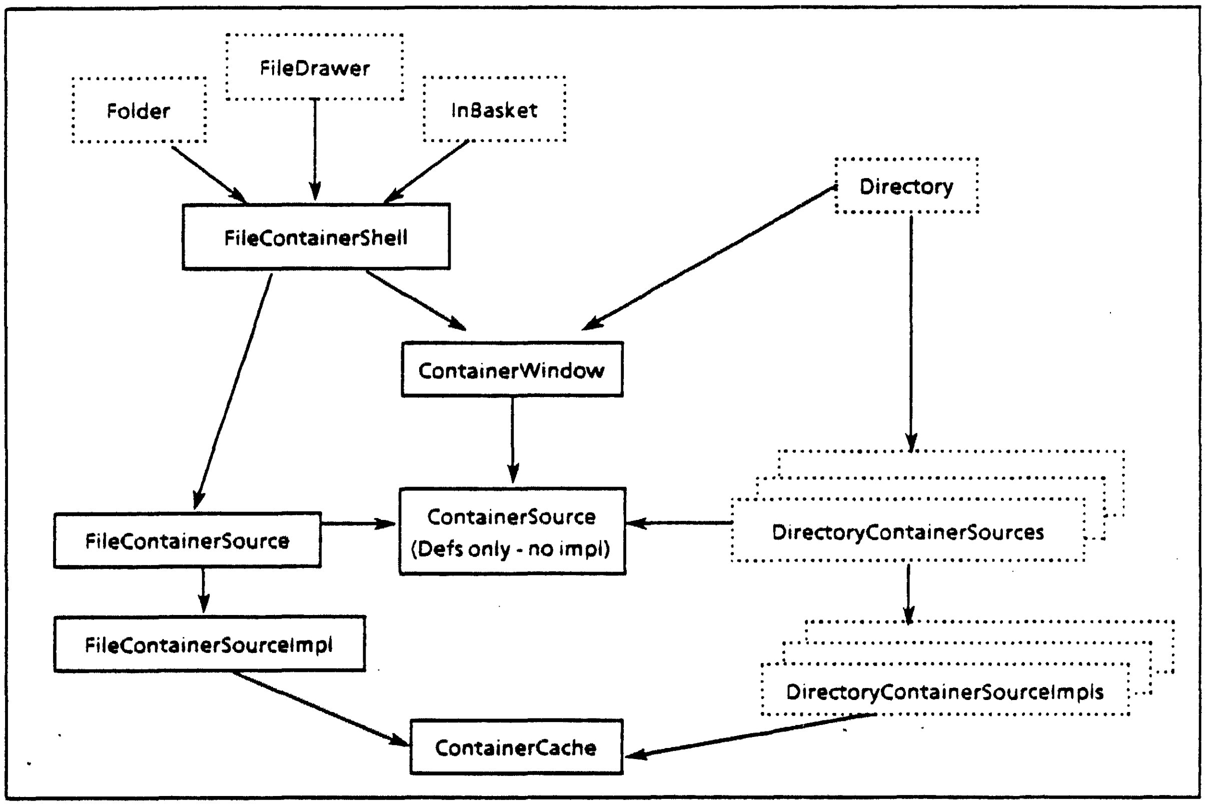


Figure 3.1 Container Interface Dependencies

The **ContainerWindow** interface takes a window and a **ContainerSource** and makes the window behave like a container. It maintains the display and manages scrolling, selection, and notifications. **Note:** This interface does not depend on **NSFile**.

A *container source* is a record of procedures that implement the behavior of the items in a container and the behavior of the container itself. **ContainerWindow** obtains the strings of each item by calling one of these procedures. **ContainerWindow** also performs user operations on items (such as open, props, delete, insert, take the current selection, and selection conversion) by calling other procedures in the record. A container source can be thought of as a supply (source) of items for a container window. The **ContainerSource** interface defines each of the procedure TYPEs that a container source must implement. **ContainerSource** contains TYPEs only.

ContainerCache provides the implementor of a container source with an easy-to-use cache for storing and retrieving the strings of each item and some client-specific data about each item.

FileContainerSource provides an **NSFile**-backed container source. It takes an **NSFile.Reference** for a file that has children, and each child file becomes an item of the container. Facilities are provided to specify the columns based on **NSFile** attributes.

The **FileContainerShell** interface takes an **NSFile** and column information (such as headings, widths, formatting) and creates a **FileContainerSource**, a **StarWindowShell**, and a container window body window. Most **NSFile**-backed container applications can use this interface, which greatly simplifes the writing of those applications.

Each of the items in a container must behave like to a file on the desktop; that is, each item must be able to be opened, show a property sheet, take a selection, and so forth. However, the items need not be backed by files. If the container is backed by an **NSFile** that has children, then the **FileContainerShell** interface is the only interface the client needs to use. Otherwise, the client must implement a container source and make most of the calls that the **FileContainerShell** implementation makes; that is, **StarWindowShell.Create**, **StarWindowShell.CreateBody**, **ContainerWindow.Create**.

3.1.10 SoftKeys

The **SoftKeys** interface provides for client-defined function keys designated to be the isolated row of function keys at the top of the physical keyboard. It also provides a **SoftKeys** window whose "keytops" may be selected with the mouse to simulate pressing the physical key on the keyboard. Such a window is displayed on the user's desktop whenever an interpretation other than the default **SoftKeys** interpretation is in effect. (The default is assumed to be the functions inscribed on the physical keys.)

The most important items in **SoftKeys** are:

Labels, LabelRecord	Strings to display on the keytops in the SoftKeys window.
Push	Install a client-specific interpretation for the soft keys.
Remove	Remove a previously installed interpretation.

3.1.11 Client-Defined Keyboards

KeyboardKey is a keyboard (the central set of black keys on the physical keyboard) registration facility. It provides clients with a means of registering system-wide keyboards (available all the time, like English, French, European), a special keyboard (like Equations), and/or client-specific keyboards (those that are available only when the client has the input focus). The labels from these registered keyboards are displayed in the softkeys window when the user holds the KEYBOARD key down.

The **BlackKeys** interface provides the data structures that define a client keyboard.

The most important items in **KeyboardKey** are:

AddToSystemKeyboards	Adds a keyboard to the system keyboards.
RegisterClientKeyboards	Establishes the keyboards available to the user.

The most important items in **BlackKeys** are:

Keyboard, KeyboardObject	A keyboard interpretation.

3.1.12 BackgroundProcess

The **BackgroundProcess** interface provides basic user feedback and control facilities to clients that want to run in a process other than the the Notifier process (see the Notifier section below). Once registered with **BackgroundProcess**, the client process can use **Attention** to post messages and check to see if the process has been aborted by the user. The user can look at the messages posted by the process and abort the process. The primary procedure in **BackgroundProcess** is **ManageMe**, which is typically the first procedure called from a background process.

3.2 Getting Started

This section is a guide for programmers who have never used the ViewPoint interfaces. It shows how two common types of applications are written using ViewPoint.

A user can invoke a program in the ViewPoint environment in two ways. First is to select an icon and press a function key such as OPEN, PROPS, COPY, or MOVE. This type of program is called an *icon application*. Second, the user may simply select an item in the attention window's pop-up menu. For example, in OS 5, a Show.Size command reports on the size of the selected icon's file. The following sections describe how to write each of these types of programs.

3.2.1 Simplest Application

The simplest way to get a program running in the ViewPoint environment is to have the program add an item to the attention window's pop-up menu. When the user selects that item, the program is called. See the SampleBWSTool for an example of this type of application. Excerpts from SampleBWSTool:

```
Init: PROCEDURE = {
   sampleTool: XString.ReaderBody ← XString.FromSTRING["Sample Tool"L];
   Attention.AddMenuItem [
      MenuData.CreateItem [
         zone: sysZ,
         name: @sampleTool,
         proc: MenuProc ] ];
   };

-- Mainline code
Init[];
```

When the application is started, its startup (mainline) code creates a MenuData.ItemHandle by calling MenuData.CreateItem and then adds this item to the attention window's menu by calling Attention.AddMenuItem. Now the MenuProc passed to MenuData.CreateItem is called when the user selects the Sample Tool item in the attention window's pop-up menu. The MenuProc can then do whatever is appropriate for the application.

3.2.2 Icon Application

Getting an icon application running in ViewPoint is a little more complex. The basic idea is that an application operates on files of a particular type. When an application is started, it registers its interest in files of that type. Whenever the user operates on a file of that type, the application gets called. Here is a skeletal example of some application code; the full explanation follows:

```
-- Constants and global data

sampleIconFileType: NSFile.Type = ...;
oldImpl, newImpl: Containee.Implementation ← [];

-- Containee.Implementation procedures

GenericProc: Containee.GenericProc = {
   SELECT atom FROM
      canYouTakeSelection = > ...
      takeSelection = > ...
      takeSelectionCopy = > ...
      open = > ...
      props = > ...
      ENDCASE = > ...

PictureProc: Containee.PictureProc = {
   ...
   Display.Bitmap [...];
   ...
   };

-- Initialization procedures

InitAtoms: PROCEDURE = {
   open ← Atom.MakeAtom["Open"L];
   props ← Atom.MakeAtom["Props"L];
   canYouTakeSelection ← Atom.MakeAtom["CanYouTakeSelection"L];
   takeSelection ← Atom.MakeAtom["TakeSelection"L];
   takeSelectionCopy ← Atom.MakeAtom["TakeSelectionCopy"L];
   };

FindOrCreatePrototypeIconFile: PROCEDURE = {...};

SetImplementation: PROCEDURE = {
   newImpl.genericProc ← GenericProc;
   newImpl.pictureProc ← PictureProc;
   oldImpl ← Containee.SetImplementation [ sampleIconFileType, newImpl ];
   };

-- Mainline code

InitAtoms[];
FindOrCreatePrototypeIconFile[];
SetImplementation[];
```

The most important thing to note in the above example is the **SetImplementation** procedure and the call to **Containee.SetImplementation** in particular. This call associates the application's implementation (**newImpl**) with a particular file type (**sampleIconFileType**). This implementation is actually a **Containee.Implementation** that is a record which contains procedures. Whenever the user operates on files of type **sampleIconFileType**, the procedures in the **Implementation** record are called. An understanding of how this works requires an understanding of how the ViewPoint desktop implementation operates.

First, some background about **NSFiles**. All **NSFiles** have:

- A name
- A file type (LONG CARDINAL)
- A set of attributes, such as create date
- Either:
  - Content, such as a document
  - Children that are also **NSFiles**, such as a folder.

An **NSFile** that has children is often called a *directory*. Fine point: an**NSFile** can actually have both content and children; however, to simplify this discussion, this point is ignored. Note: Because the children of an **NSFile** can themselves have children, **NSFile** supports a hierarchical file system.

A ViewPoint desktop is an **NSFile** that has children. An on-screen icon picture represents each child file of the desktop's **NSFile** The desktop display of rows of "icons" is an illusion. The word *icon* is in quotes because, from the programmer's point of view, there really is no such thing as an icon. The only things that really exist are files (**NSFiles**), icon *pictures*, and application code.

Immediately after logging on, the desktop implementation enumerates the child files of the desktop file and calls an application's **Containee.PictureProc** for each child file, based on the child file's type. Each application's **Containee.PictureProc** should then display the icon picture for that file.

After logon is complete and the desktop is displayed, the desktop implementation receives user actions such as mouse clicks and presses of the OPEN or PROPS keys. For example, assume the user selects an icon picture and presses OPEN. The desktop implementation determines the file type for the file represented by the icon picture the user selected and then calls the **Containee.GenericProc** for the application that operates on files of that type, requesting that the application open the icon. It also passes the application a unique identifier for the particular file selected. At this point, the application can do whatever is appropriate for that application. Typically, the application opens the file, reads some data out of it, creates a **StarWindowShell**, and displays the contents of the file in the window in some application-specific form.

The desktop implementation does not call an application directly. Rather, ViewPoint maintains a table of file-type/**Containee.Implementation** pairs. When an application calls **Containee.SetImplementation**, an entry is added to the table. When the desktop implementation calls an application, it obtains the Containee.Implementation for the application by looking it up in the table (it actually calls Containee.GetImplementation).

3.2.3 Operational Notes

To write an icon application, a programmer *must obtain a unique file type*. Contact your ViewPoint consultant to obtain one.

In the example above, the application in its initialization code checks to be sure a prototype file exists and, if not, creates one. This usually involves creating a file with the proper file type for this application. This allows the user to get started with the application, usually by copying the blank prototype out of a special folder of prototypes.

Note: There is a clear distinction between a prototype file for an application and a bcd file that contains the code for the application. All bcd files are of the same type, while each prototype file is different for each application.

3.3 Flow Descriptions

The following flow descriptions are intended to show how everything is related. For each example scenario, the exact sequence of calls is described, including ViewPoint interfaces and clients.

3.3.1 Select an Icon

The user points at an icon on the desktop.

- When the mouse button goes down over an icon picture, the notification goes to the desktop implementation's TIP.NotifyProc. The NotifyProc will be passed a Window.Place and a "PointDown" atom. The desktop implementation determines what file is represented by that icon picture. Fine point: The desktop implementation maintains a mapping from icon picture locations to NSFile.References.
- The desktop implementation calls Containee.GetImplementation, passing in the file type of the file and getting back the Containee.Implementation for that file type.
- The desktop implementation calls the Containee.PictureProc that is in the Implementation; (that is, impl.pictureProc), passing in:
    - data: the NSFile.Reference for the file
    - old: normai
    - new: highlighted
- The application's PictureProc displays a highlighted version of its icon picture, perhaps simply calling Display.Invert.
- When the mouse button goes up (a "PointUp" atom), the desktop implementation becomes the current selection manager by calling Selection.Set. It sets the desktop window to be the current input focus by calling TIP.SetInputFocus. Setting the input

focus to be the desktop window ensures that keys such as OPEN, PROPS, COPY,and so forth, will all go to the desktop's NotifyProc.

- END.

3.3.2 PROPS of an Icon

Assume an icon on the desktop is selected. The user presses PROPS. After changing some items in the property sheet, the user selects Done.

- The desktop implementation's TIP.NotifyProc gets the notification (a "PropsDown" atom) and determines which icon picture is currently selected and what file is represented by that icon picture.

- The desktop implementation calls Containee.GetImplementation, passing in the file type of the file and getting back the Containee.Implementation for that file type.

- The desktop implementation calls the Containee.GenericProc that is in the Implementation; (that is, impl.genericProc), passing in:
  - data: the NSFile.Reference for the file
  - atom: "Props"
  - changeProc: a Containee.ChangeProc that belongs to the desktop implementation
  - changeProcData: a pointer to some desktop implementation data that identifies the icon/file being operated on.

- The application's GenericProc creates a property sheet by calling PropertySheet.Create. It probably also opens and retrieves some data out of the file (using various NSFile operations) and uses that data to set the initial values of the items in the property sheet.

- Typically, the client wants to save the NSFile.Handle for the file while the property sheet is open. In addition, if the opening and closing of the property sheet might cause the file's attributes to change, the application's GenericProc must save the passed changeProc and changeProcData. A typical example is when the file's name is one of the items in the property sheet and the user can change the name. The data is saved by allocating a record with this data in it and passing a pointer to the record as the clientData parameter to PropertySheet.Create. Later, when the user selects Done or Apply, this data may be recovered (see the rest of this flow description). **Note:** This data cannot be saved in a local frame (such as that of the GenericProc) because the GenericProc must *return* to the notifier after creating the property sheet; when the user selects Done or Apply that is a new call stack. The client data should not be saved in a global frame because more than one property sheet may be open for a particular application.

- The application's GenericProc returns the StarWindowShell.Handle for the property sheet.

- The desktop implementation displays the property sheet by calling StarWindowShell.Push; then the desktop's NotifyProc returns to the Notifier.

- The user changes some items and then selects Done.
- The **PropertySheet** implementation calls the client's **PropertySheet.MenuItemProc** that was passed in to **PropertySheet.Create**, passing in:
  - **shell**: the **StarWindowShell** for the property sheet
  - **formWindow**: the **FormWindow** for the property sheet
  - **menuItem: done**
  - **clientData**: the pointer to the client's data that was passed to **PropertySheet.Create**.
- The client's **MenuItemProc** recovers the client's data (the file handle, the **changeProc** and **changeProcData**, and any other relevant client data) from the **clientData** parameter. It determines if the user made any changes and, if so, updates the file accordingly *and* calls the **changeProc**, passing in the **changeProcData**, the file reference, and a list of the changed file attributes.
- The desktop's **ChangeProc** causes the icon picture to be redisplayed, because changing an attribute such as the name requires the picture to be updated with the new name.
- The client's **MenuItemProc** returns to the **PropertySheet** implementation, indicating that the property sheet should be destroyed.
- The **PropertySheet** implementation destroys the property sheet by calling **StarWindowShell.Pop** and returns to the Notifier.
- END.

3.3.3 OPEN an Icon

Opening an icon is similar to opening a property sheet for an icon.

3.3.4 COPY Something to an Icon

Assume something has been selected. The user presses COPY and then points at an icon.

- When the user presses COPY, the **NotifyProc** for the window that currently has the input focus (and the selection) is called. It calls **TIPStar.SetMode [copy]** to set the environment into copy mode and then returns to the Notifier. It might also call **Cursor.Set** to change the cursor shape to indicate move mode.
- **SetMode** replaces the **NormalMouse.TIP** table with the **CopyModeMouse.TIP** table.
- The user presses the mouse button down over an icon on the desktop.
- The desktop's **NotifyProc** gets called with a "CopyModeDown" atom (instead of a "PointDown" atom because of the TIP table switch). It determines what file is represented by the icon picture the user is pointing at. It calls **Containee.GetImplementation**, passing in the file's type and getting back a **Containee.Implementation**. It calls the **Implementation**'s **GenericProc** passing in:
  - **data**: the **NSFile.Reference** for the file

- atom: "CanYouTake"

- The application's GenericProc calls Selection.CanYouConvert or Selection.HowHard to determine if the current selection can be converted to target type(s) that the application can take. For example, if the icon being copied to is a printer icon, it calls HowHard with targets of interpressMaster and file.

- The Selection implementation calls the current selection manager's Selection.ConvertProc. It returns an indication of how hard it would be to convert the selection to the given target types.

- The application's GenericProc returns a pointer to TRUE if it determines that it can take the current selection and FALSE if it cannot.

- The desktop implementation changes the cursor shape to a question mark if the application's GenericProc returns FALSE. Otherwise, it leaves the cursor as it was.

- The user releases the mouse button.

- The desktop's NotifyProc gets called with a "CopyModeUp" atom. It determines what file is represented by the icon picture the user is pointing at. It calls Containee.GetImplementation, passing in the file's type and getting back a Containee.Implementation. It then calls the Implementation's GenericProc, passing in:

  - data: the NSFile.Reference for the file
  - atom: "TakeSelectionCopy"
  - changeProc: a Containee.ChangeProc that belongs to the desktop implementation
  - changeProcData: a pointer to some desktop implementation data that identifies the icon/file being copied to

- The application's GenericProc calls Selection.Convert or (Selection.Enumerate) to convert the selection to the desired type. The application then operates on the converted selection value as appropriate for that application. For example, the printer icon application converts the selection to an interpressMaster and sends the master to the printer. (See the Selection chapter for a full flow description of the selection mechanism.)

- The application's GenericProc returns to the desktop's NotifyProc, which returns to the Notifier.

- END.

3.4 Programming Conventions

The ViewPoint environment assumes that the programs that run in it are friendly and that they are not trying to circumvent or sabotage the system. The system does not enforce many of the conventions described here but assumes that application programmers will adhere to them voluntarily. If these conventions are not followed, the ViewPoint environment may degrade or break down altogether.

The most important principle is that users should have complete control over their environment. In particular, clients shall not pre-empt users. A user should never be forced by a client into a situation where the only thing that can be done is to interact with only one application. Furthermore, the client should avoid falling into a particular mode when interacting with the user; that is, an application should avoid imposing unnecessary restrictions on the sequence of user actions.

This goal of user control has implications for the designs of applications. A client should never seize control of the processor while getting user input. This tends to happen when the client wants to use the "get a command from the user and execute it" mode of operation. Instead, an application should arrange for ViewPoint to notify it when the user wishes to communicate some event to the application. This is known as the "Don't call us, we'll call you" principle.

The user owns the window layout on the screen. Although the client can rearrange the windows, this is discouraged. Users have particular and differing tastes in the way they wish to lay out windows on the display; it is not the client's role to override the user's decisions. In particular, clients should avoid making windows jump up and down to try to capture the user's attention. If the user has put a window off to the side, then he does not want to be bothered by it.

3.4.1 Notifier

ViewPoint sends most user input actions to the window that has set itself to be the focus for user input; the rest of the actions are directed to the window containing the cursor. (See the **TIP** interface for details on how the decision is made where to send these actions.) A process in ViewPoint notes all user input actions and determines which window should receive each one. A client is concerned only with the actions that are directed to its window; it need not concern itself with determining which actions are intended for it.

The basic notification mechanism directs user input to one of many windows in the window tree. Each window has a **TIP.Table** and a **TIP.NotifyProc**. The table is a structure that translates a sequence of user actions into a sequence of results that are then passed to the notify procedure of the window.

There are two processes that share the notification responsibilities, the Stimulus process and the Notifier process. The Stimulus process is a high-priority process that wakes up approximately 50 times a second. When it runs, it makes the cursor follow the mouse and watches for keyboard keys going up or down, mouse motion, and mouse buttons going up or down, enqueuing these events for the Notifier process.

The Notifier process dequeues these events, determines which window the event is for, and tries to match the events in the window's table. If it finds a match in the table, it calls the window's notify procedure with the results specified in the table. If no match is found, it tries the next table in the window's chain of tables. If no match is found in any table, the event is discarded.

The Notifier process is important. To avoid multi-process interference, some operations in the system can happen only in the Notifier process. Setting the selection is one such operation. The Notifier process is also the one most closely tied to the user. The Notifier waits until a NotifyProc finishes for one user action before processing the next user action. If an operation takes an extended time to complete (more than three to five seconds), it should be forked from the Notifier process to run in a separate process so that the Notifier process is free to respond to the user's actions. Of course, the application writer must take great care when stepping into this world of parallel processing.

3.4.2 Multiple Processes, Multiple Instances

In ViewPoint, many programs can run simultaneously. The designer of a client-callable package should bear in mind that several different asynchronous clients may invoke his package, so the package should be monitored.

The simplest design is to have a single entry procedure that all clients must call. While one client is using the package, all other clients block on the monitor lock. Of course, no state should be maintained internally between successive calls to the package, because there is no guarantee that the same client is calling each time.

This simple approach has the disadvantage that clients are simply stopped for what may be a long time, with no option of taking alternate action. To ease this restriction, the entry procedure can check a "busy" bit in the package. If the package is busy, the procedure can return this result to the client. The client can then decide whether to give up, try something else, or try again. This is less likely to tie up an application for a long period, and the user can use the application for other purposes.

If the package is providing a collection of procedures and cannot provide its services in a single procedure, the package and its clients must pass state back and forth in the form of an object. The package can use a single monitor on its code to protect the object, or it can provide a monitor as part of each object. If it does the latter, then several clients can be executing safely at the same time.

Some packages require that a client provide procedures that are called by the package. The designer of such a package should have these client-provided procedures take an extra parameter, a long pointer to client instance data. When the client provides the package with the procedures, it also provides the instance data to pass to the procedures when they are called. The client can then use this instance data to distinguish between several different instances of itself that are sharing the same code.

3.4.3 Resource Management

Programs in the Xerox Development Environment must explicitly manage resources. For example, memory is explicitly allocated and deallocated by programs; there is no garbage collector to reclaim unused memory. All programs share the same pool of resources, and there is no scheduler watching for programs using more than their share of execution time, memory, or any other resource.

Programs must manage resources carefully. If a program does not return a resource when it is done with it, that resource will never become available to any other program and the performance of the environment will degrade. The most common resource, and one of the more difficult to manage, is memory.

When interfaces exchange resources, clients must be very careful about who is responsible for the resource. The program that is responsible for the deallocation of a resource is the *owner* of that resource. One example of a resource is a file handle. If a program passes a file handle to another program, both programs must agree about who owns that file handle. Did the caller transfer ownership by passing the file handle, or is it retaining ownership and only letting the called procedure use the file handle? If there is disagreement between the two programs, either the file will be released twice, or it will never be released at all. All interfaces involving resources must state explicitly whether ownership is transferred. To ease the problem of memory management when the ownership of memory can change, a heap called the *system heap* is used in ViewPoint. If a piece of memory can have its ownership transferred, it is either allocated from the system heap or a deallocation procedure must be provided for it.

The most common resource appearing in interfaces is an **XString** (**Reader** or **ReaderBody**). There must be agreement about which program is responsible for deallocating the string's bytes. Typically, a string passed as an input parameter does not carry ownership with it; implementors of such procedures should not deallocate or change the string. If it is necessary for the implementor to modify the string or use it after the procedure returns, the implementor should first copy it. Clients should be particularly careful when a procedure returns a string to note whether ownership has come with it.

3.4.4 Stopping Applications

The ViewPoint environment consists of cooperating processes. There are no facilities for cleanly terminating an arbitrary collection of processes. It is assumed that application writers are good citizens and will design their tools to stop voluntarily when asked to stop.

An application should stop if the user aborts the application. There are two ways to determine if the user has aborted an application. (1) An application's window can have a **TIP.AttentionProc** that is called as soon as the user presses the STOP key. (2) Procedures in the **TIP** interface can check whether a user has aborted an application with the STOP key in the application's window. An application should check for a user abort at frequent intervals and be prepared to stop executing and clean up after itself. Because the application controls when it checks, it can check at points in its execution when its state is easy to clean up. Packages that can be called from several programs should take a procedure parameter that can be called to see whether the user has aborted.

3.4.5 Multinationality

ViewPoint is designed to support easy transport of applications to other countries. The string package (**XString**, **XChar**, **XFormat**, and so forth) supports the *Xerox Character Code Standard*, which allows for strings in many languages to be intermixed. The **XMessage** interface allows user messages to be translated into other languages because the application programmer can put all these messages into a module separate from the rest of the application code. The **KeyboardKey** interface supports the addition of keyboards for many languages.

Application programmers are strongly encouraged to allow their application to be multilingual. This means for example, using **XString** for all string operations and using **XMessage** to manage any text that will be displayed to the user. It also means not making any language assumptions about characters received from the user. An application that expects typing input from the user should be prepared to receive characters from *any* character set.

3.5 Summary of Interfaces

Atom provides the mechanism for making **TIP**, **Event**, and **Containee** atoms.

AtomicProfile provides a mechanism for storing and retrieving global values.

Attention provides a means of displaying messages to the user.

BackgroundProcess provides basic user feedback and control facilities to clients that want to run in a process other than the the Notifier process.

BlackKeys provides the capability to change the interpretation of the central (black) section of the keyboard.

Containee is an application registration facility. It allows an application to register its implementation for files of a particular type.

ContainerCache provides a simple cacheing mechanism for the implementor of a container source.

ContainerSource defines the procedures that must be implemented to provide a source of items for a container window.

ContainerWindow creates a window that displays an ordered list of items that behave like icons on a desktop.

Context provides a mechanism for clients to associate data with windows.

Cursor provides facilities for a client to manipulate the appearance of the cursor that represents the mouse position on the screen.

Display provides facilities to display bits in windows.

Event provides clients with the ability to be notified of events that take place asynchronously on a system-wide basis.

FileContainerShell creates a **StarWindowShell** with a **ContainerWindow** as a body window that is backed by a **FileContainerSource**.

FileContainerSource creates a container source that is backed by a file that has children.

FormWindow creates a window with various types of form items in it, such as text, boolean, choice (enumerated), command, and window. **FormWindow** is used to create property sheets.

FormWindowMessageParse provides procedures that parse strings to produce various **FormWindow** TYPEs.

IdleControl provides access to the basic controlling module of ViewPoint.

KeyboardKey is a client keyboard (the central black keys) registration facility.

KeyboardWindow provides a particular implementation for a keyboard window.

LevelIVKeys defines the names of the physical keys.

MenuData allows menus and menu items to be created.

MessageWindow provides a facility for posting messages in a window to the user .

PopupMenu allows a menu to be displayed (popped up) anywhere on the screen.

PropertySheet creates a property sheet. A property sheet shows the properties of some object to the user and allows the user to change the properties.

Selection provides the facilities for a client to manipulate the user's current selection. It also provides procedures that enable someone other than the originator of the selection to request information relating to the selection and to negotiate for a copy of the selection in a particular format.

SimpleTextDisplay provides facilities for displaying, measuring, and resolving strings of Xerox Character Code Standard text. It can handle only nonattributed single-font text.

SimpleTextEdit provides facilities for presenting short, editable pieces of text to the user.

SimpleTextFont provides access to the default system font that is used to display ViewPoint's text, such as the text in menus, the attention window, window names, containers, property sheet text items, and so forth.

SoftKeys provides for client-defined function keys designated to be the isolated row of function keys at the top of the physical keyboard.

StarDesktop provides access to the user's desktop file and window.

StarWindowShell provides facilities for creating Star-like windows.

TIP provides basic user input facilities through a flexible mechanism that translates hardware level actions from the keyboard and mouse into higher-level client action requests.

TIPStar provides access to ViewPoint's normal set of **TIP** tables.

Undo provides facilities that allow an application to register undo opportunities, so that when the user requests that something be undone, the application is called to do so.

Window defines the low-level window management package used by ViewPoint.

XChar defines the basic character type as defined in the *Xerox Character Code Standard* as well as some operations on it.

XCharSetNNN enumerates the character codes in character set NNN.

XCharSets enumerates the character sets defined in the *Xerox Character Code Standard.*

XComSoftMessage defines messages for some commonly used strings, such as Yes, No, day-of-the-week, month,and so forth.

XFormat converts various TYPEs into **XStrings**.

XLReal supports manipulation of real numbers with greater precision than Mesa REALs.

XMessage supports the multilingual requirements of systems that require the text displayed to the user be separable from the code and algorithms that use it.

XString provides the basic data structures for representing encoded sequences of characters as defined in the *Xerox Character Code Standard*. It also provides several operations on these data structures.

XTime provides facilities to acquire and edit times into **XStrings** and **XStrings** into times.

XToken parses **XStrings** into other TYPES

4

AdjustableWindow

4.1 Overview

AdjustableWindow makes an arbitrary window become shrinkable, growable, and/or moveable by the user. For a comprehensive overview of all the subwindow interfaces and their intended use, see the Subwindow Overview chapter.

4.2 Interface Items

4.2.1 Create Proc

```
Create: PROC [
    window: Window.Handle,
    adjustProc: AdjustProc,
    zone: UNCOUNTED ZONE,
    limitProc: LimitProc ← NIL,
    adjustableEdges: Edges ← defaultAdjustableEdges,
    topBottom, move: BOOLEAN ← FALSE,
    upperCornersColor, lowerCornersColor: Display.Brick ← NIL -- NIL means Gray--]

Edge: TYPE = {left, right, top, bottom};
Edges: TYPE = PACKED ARRAY Edge OF BooleanFalseDefault;
BooleanFalseDefault: TYPE = BOOLEAN ← FALSE;
defaultAdjustableEdges: Edges = [
    left: FALSE, right: TRUE, top: FALSE, bottom: TRUE];
```

Create makes window resizeable, moveable and/or top/bottomable. adjustableEdges governs which edges or corners can be adjusted. defaultAdjustableEdges, for example, means the lower right corner of window is adjustable (which means that the upper left corner is fixed). If only bottom is TRUE the only possible resize is a "drag" of the bottom edge of the window. The window is moveable if move is TRUE and top/bottomable if topBottom is TRUE. upperCornersColor and lowerCornersColor allow the client to specify white or gray etc. for the adjustable quadrants. adjustProc and limitProc are the adjust and limit procs for window.

4.2.2 Adjust and Limit Procs

```
AdjustProc: TYPE = PROCEDURE [
    window: Window.Handle,
    box: Window.Box,
    when: When];
```

```
When: TYPE = {before, after};
```

An AdjustProc is the proc that is called when window's box is changing. It will be called both before and after the Window.SlideAndSize occurs.

```
GetAdjustProc: PROC [window: Window.Handle]
    RETURNS [AdjustProc];
```

Get the AdjustProc associated with window.

```
SetAdjustProc: PROC [
    window: Window.Handle,
    proc: AdjustProc]
RETURNS [old: AdjustProc];
```

Set the AdjustProc for window to be proc. Returns the previously set AdjustProc.

```
LimitProc: TYPE = PROCEDURE [
    window: Window.Handle,
     box: Window.Box]
    RETURNS [newBox: Window.Box];
```

A LimitProc is the proc that is called when window box is about to change. It allows the client a chance to disallow (newBox ← oldBox) or modify (newBox ← otherBox) the proposed new box value before the call to Window.SlideAndSize occurs.

```
GetLimitProc: PROC [window: Window.Handle]
    RETURNS [LimitProc];
```

Get the LimitProc associated with window.

```
SetLimitProc: PROC [
    window: Window.Handle,
    proc: LimitProc]
RETURNS [old: LimitProc];
```

Set the LimitProc for window to be proc. Returns the previously set LimitProc.

4.2.3 Utilities

```
IsIt: PROCEDURE [window: Window.Handle] RETURNS [yes: BOOLEAN];
```

Returns TRUE if window is an adjustable window.

```
GetZone: PROCEDURE[window: Window.Handle] RETURNS [zone: UNCOUNTED ZONE];
```

Returns the zone associated with window.

4.2.4 Friends

```
CallAncestorForTheseCorners: PROCEDURE [
    window: Window.Handle,
    corners: Corners ← ALL [FALSE]];

Corner: TYPE = {upperLeft, upperRight, lowerLeft, lowerRight};
Corners: TYPE = PACKED ARRAY Corner OF BooleanFalseDefault;
```

CallAncestorForTheseCorners should be called if the client wishes parent windows to receive notifications (like the shell being notified when adjustments are made to the bottom subwindow in a subdivided shell). For example, a subwindow can be designed to fill the lower segment of a shell and it's lower grabbers to cause the shell to resize as well as the subwindow. SubwindowManager uses this technique.

4.2.5 Errors

```
Error: ERROR [type: ErrorType];

ErrorType: TYPE = {notAnAdjustableWindow, notAllowed};
```

4.3 Usage/Examples

```
--from SubwindowManager.MakeSW
--make the new sw adjustable and set the manager procs
AdjustableWindow.Create[sw, CoordinateSWAdjusts, zone,
CoordinateSWLimits, [FALSE,FALSE,horizAdjust,vertAdjust], FALSE, FALSE];
AdjustableWindow.CallAncestorForTheseCorners[
    sw, [FALSE,FALSE,TRUE,TRUE]];
IF SubwindowFriends.GetSWProcs[type].scrollSWProc # NIL THEN
    sw ← SubwindowFriends.GetSWProcs[type].scrollSWProc[
    sw, vertScrollbar, horizScrollbar];
```

4.4 Index of Interface Items

Item	Page
AdjustProc: TYPE	2
BooleanFalseDefault: TYPE	1
CallAncestorForThese Corners: PROCEDURE	3
Corner: TYPE	3
Corners: TYPE	3
Create: PROCEDURE	1
defaultAdjustableEdges: Edges	1
Edge: TYPE	1
Edges: TYPE	1
Error	3
ErrorType: TYPE	3
GetAdjustProc: PROCEDURE	2
GetLimitProc: PROCEDURE	2
GetZone: PROCEDURE	3
IsIt:PROCEDURE	2
LimitProc: TYPE	2
SetAdjustProc: PROCEDURE	2
SetLimitProc: PROCEDURE	2
When: TYPE	2

5 ApplicationFolder

5.1 Overview

ApplicationFolder provides access to the folder that contains all the component files of an application. A full application is composed of one or more bcds, a message file, a description file, and other data files such as .TIP or .Icons. These components are all put together into a folder with a specific file type, called an Application (or ApplicationFolder).

When the application is loaded and started, one of the first things it does is get its data files. The actual file names of the data files are usually specified in the application's description file, which is a file that may be read using the OptionFile interface. The application gets its data files by using ApplicationFolder.FromName to obtain the ApplicationFolder file, using ApplicationFolder.FindDescriptionFile to get the description file from the ApplicationFolder file, and then using OptionFile.GetStringValue to get the data files names. (See Usage/Examples.) ApplicationFolderExtra.InitMessages makes initializing messages much easier, just pass in an internal name and get back an XMessage.Handle. No more looking for description file, finding the message file, etc.

5.2 Interface Items

```
FromName: PROCEDURE [internalName: XString.Reader]
    RETURNS [applicationFolder: NSFile.Reference];
```

Returns the folder for the given application. internalName is the section name in the description file. Returns NSFile.nullReference if not found.

```
FindDescriptionFile: PROCEDURE [applicationFolder: NSFile.Handle]
    RETURNS [descriptionFile: NSFile.Reference];
```

Finds a file with file type = OptionFile (4385) in the applicationFolder. Returns NSFile.nullReference if not found.

```
FindDescriptionFileX: PROCEDURE [applicationFolder: NSFile.Handle, session: NSFile.Session]
    RETURNS [descriptionFile: NSFile.Reference];
```

Finds a file with file type = **OptionFile** (4385) in the **applicationFolder** using the specified **NSFile.Session**. Returns **NSFile.nullReference** if not found. Fine Point: In BWS4.3, this is in ApplicationFolderExtra.mesa.

```
EventData: TYPE = RECORD [
    applicationFolder: NSFile.Reference,
    internalName: XString.Reader,
    applicationADF: NSFile.Reference,
    containsFontFile: BOOLEAN,
    versionStamp: CARDINAL,
    priority: CARDINAL];
```

```
versionStamp: CARDINAL = 2;
```

The application loader notifies the **ApplicationLoaded** event after loading and starting an application. **EventData** is passed as **Event.EventData** for this event. **applicationADF** is the description file, **containsFontFile** is TRUE if it contains **NovaFontFile** entry in the description file. The current **versionStamp** is 2. If the **versionStamp** is 2, then the priority is cached as an extended attribute of the application folder. The value of **priority** of an application folder is specified in the ADF's priority entry. At startup time, the autorun applications are started in the order of ascending priority number. Fine Point: In BWS4.3, EventData is in ApplicationFolderExtra2.mesa.

The application loader also notifies the following events:

AboutLoading: Notifies when the loader try to load an application.
LoadVetoed: Notifies when loading vetoed by the client of the **AboutLoading** event.
LoadedAndAboutToStart: Notifies when finished loading an application.

```
InitMessages: PROCEDURE [internalName: XString.Reader, label: XString.Reader ← NIL,
    domainindex: CARDINAL ← 0] RETURNS [h: XMessage.Handle] ;
```

Returns initialized **XMessage.Handle** for the specified application folder, **internalName** is the section name in the description file. If **label** is non-NIL, then **label** is used as the entry name in the description file. If **label** is NIL, then the entry **MessageFile** is used. **domainindex** is **XMessage.MsgDomains**. Fine Point: In BWS4.3, this is in ApplicationFolderExtra.mesa.

5.3 Usage/Examples

This example code obtains the message file.

```
-- File: SampleMsgFileInitImpl.mesa - last edit:

-- Copyright (C) 1985 by Xerox Corporation. All rights reserved.

DIRECTORY
    ApplicationFolder USING [FindDescriptionFile, FromName],
    Heap USING [systemZone],
    NSFile USING [Close, Error, GetReference, Handle, nullHandle, nullReference, OpenByNan
    OpenByReference, Reference, Type],
    NSString USING [FreeString, String],
    OptionFile USING [GetStringValue],
    SampleBWSApplicationOps,
    XMessage USING [ClientData, FreeMsgDomainsStorage, Handle, MessagesFromReference,
    MsgDomains],
    XString USING [FromSTRING, NSStringFromReader, Reader, ReaderBody];

SampleMsgFileImpl: PROGRAM
    IMPORTS ApplicationFolder, Heap, NSFile, NSString, OptionFile, XMessage, XString
    EXPORTS SampleBWSApplicationOps = {

-- Data

h: XMessage.Handle ← NIL;

localZone: UNCOUNTED ZONE ← Heap.systemZone;

-- Procedures

DeleteMessages: PROCEDURE [clientData: XMessage.ClientData] = {};

GetMessageHandle: PUBLIC PROCEDURE RETURNS [XMessage.Handle] = {RETURN[h]};

InitMessages: PROCEDURE = {
    internalName: XString.ReaderBody ← XString.FromSTRING ["SampleBWSApplication"L];
    msgDomains: XMessage.MsgDomains ← NIL;
    msgDomains ← XMessage.MessagesFromReference [
        file: GetMessageFileRef [ApplicationFolder.FromName [@internalName]],
        clientData: NIL,
        proc: DeleteMessages ];
    h ← msgDomains[0].handle;
    XMessage.FreeMsgDomainsStorage [msgDomains];
    };

GetMessageFileRef: PROCEDURE [folder: NSFile.Reference]
    RETURNS [msgFile: NSFile.Reference ← NSFile.nullReference] = {
    folderHandle: NSFile.Handle ← NSFile.OpenByReference [folder];
    internalName: XString.ReaderBody ← XString.FromSTRING ["SampleBWSApplication"L];
    messageFile: XString.ReaderBody ← XString.FromSTRING ["MessageFile"L];

    FindMessageFileFromName: PROCEDURE [value: XString.Reader] = {
        nssName: NSString.String ← XString.NSStringFromReader [r: value, z: localZone];
        msgFileHandle: NSFile.Handle ← NSFile.nullHandle;
        msgFileHandle ← NSFile.OpenByName [directory: folderHandle, path: nssName !
            NSFile.Error = > {msgFileHandle ← NSFile.nullHandle; CONTINUE}];
        IF msgFileHandle = NSFile.nullHandle THEN ERROR; -- no message file!
        msgFile ← NSFile.GetReference [msgFileHandle];
        NSFile.Close [msgFileHandle];
        NSString.FreeString [z: localZone, s: nssName];
        };

    OptionFile.GetStringValue [section: @internalName, entry: @messageFile,
        callBack: FindMessageFileFromName,
        file: ApplicationFolder.FindDescriptionFile [folderHandle]];
    NSFile.Close [folderHandle];
    };

-- Mainline code

InitMessages[];

}...
```

5.4 Index of Interface Items

6

Atom

6.1 Overview

Although it is often convenient to name an object using a textual name, XStrings are somewhat clumsy to compare and pass around. An *atom* is a one-word datum that has a one-to-one correspondence with a textual name. Using atoms, objects may be named textually without having to store, copy, and compare the strings themselves. Atoms were made popular by the Lisp language.

The textual name associated with an atom is called its PName, just as it is in Lisp. If two atoms are equal, they correspond to the same PName and vice versa. An atom may also have properties associated with it; a property is a [name, value] pair.

6.2 Interface Items

6.2.1 Making Atoms

ATOM: TYPE[1];

null: ATOM = LOOPHOLE[0].

An ATOM is a one-word datum that has a one-to-one correspondence with a textual name, or PName. If two ATOMs are equal, they correspond to the same pName. If two pNames are equal, they correspond to the same ATOM.

Make: PROCEDURE [pName: XString.Reader] RETURNS [atom: ATOM];

MakeAtom: PROCEDURE [pName: LONG STRING] RETURNS [atom: ATOM];

MakeAtom and Make return the ATOM corresponding to pName, creating one if necessary. In pName, uppercase and lowercase characters are different and result in different ATOMs. The atom returned is valid for the duration of the boot session, and the pName is remembered for the duration of the boot session.

GetPName: PROCEDURE [atom: ATOM] RETURNS [pName: XString.Reader];

GetPName returns the name of **atom**, returning NIL if **atom** is **null**. It raises the error **NoSuchAtom** if **atom** is not valid.

6.2.2 Error

```
NoSuchAtom: ERROR;
```

NoSuchAtom may be raised by **GetPName**, **PutProp**, **GetProp**, or **RemoveProp**. It is raised when an operation is presented with an **ATOM** for which no **Make** or **MakeAtom** operation has been done in the boot session. Such atoms are called *invalid atoms*.

6.2.3 Property Lists

```
Pair: TYPE = RECORD [prop: ATOM, value: RefAny];
```

```
RefAny: TYPE = LONG POINTER;
```

```
RefPair: TYPE = LONG POINTER TO READONLY Pair;
```

Pair defines the [name, value] pair for a property. Properties are named by atoms and have long pointers as values. Property pairs are referenced by a read-only pointer.

```
PutProp: PROCEDURE [onto: ATOM, pair: Pair];
```

PutProp adds a property pair to **onto**. If the property already exists, the value is updated. If **onto** is **null**, no action takes place. **PutProp** raises the error **NoSuchAtom** if **onto** is not valid.

```
GetProp: PROCEDURE [onto, prop: ATOM] RETURNS [pair: RefPair];
```

GetProp returns the property pair whose property name is the atom **prop** from atom **onto**. If **onto** does not have a property whose name is **prop** or **onto** is **null**, NIL is returned. **GetProp** raises the error **NoSuchAtom** if **onto** is not valid. **Note:** The client may not change the property pair.

```
RemoveProp: PROCEDURE [onto, prop: ATOM];
```

RemoveProp removes the property pair whose property name is the atom **prop** from atom **onto**. If **onto** is **null**, no action takes place. **RemoveProp** raises the error **NoSuchAtom** if **onto** is not valid.

6.2.4 Enumerating Atoms and Property Lists

```
MapAtomProc: TYPE = PROCEDURE [ATOM] RETURNS [BOOLEAN];
```

MapAtomProc is used by **MapAtom** to enumerate atoms. When it returns TRUE, the enumeration stops.

```
MapAtoms: PROCEDURE [proc: MapAtomProc] RETURNS [lastAtom: ATOM];
```

MapAtoms enumerates the atoms, calling proc once for each atom. If proc returns TRUE, MapAtoms returns that atom. If proc never returns TRUE, MapAtoms returns null.

```
MapPListProc: TYPE = PROCEDURE [RefPair] RETURNS [BOOLEAN];
```

MapPListProc is used by MapPList to enumerate property lists. When it returns TRUE, the enumeration stops. **Note:** The client may not change the property pair.

```
MapPList: PROCEDURE [atom: ATOM, proc: MapPListProc] RETURNS [lastPair: RefPair];
```

MapPList enumerates the property list of atom, calling proc once for each pair. If proc returns TRUE, MapPList returns that pair. If proc never returns TRUE, MapPList returns NIL.

6.3 Usage/Examples

Atom is most appropriately used for communicating names and permanent data between separate applications or between far-flung parts of a single application. The AtomicProfile interface is an example of this use.

However, ATOMs and atom property lists add to the working set of every application, and thus degrade system performance as a whole. This happens because Atom must make a copy of the atom name in its (permanent) database, and every client of Atom uses that database. It is much better to keep an application's data separated from other data.

Property lists are a shared, global resource and should be used for sharing other global resources. They should not be used for transient data. For example, consider the chaos that would ensue if several instances of an application were running simultaneously and each assumed that the property list of a particular atom was its to read and write. (Of course, this interference could also result from different applications running at different times.)

ATOMs take a significant amount of time to create. Applications interested in good performance will only use ATOMs if they need a runtime-extendable enumeration; a simple compile-time enumeration is much more efficient.

If you want an atom with a property list for a private or transient usage (a bad idea in any case) you must make sure that the atom is unique, so as not to interfere with other applications using the same atom. Code such as

```
   myList: Atom.ATOM = Atom.MakeAtom["string list"L]; -- WRONG
```

must be replaced by code that gives an atom name that is unique to the application or module (or instance, if multiple instances may be running).

Two of the major uses of atoms are in the Event and TIP interfaces. In the Event interface, atoms name events. In the TIP interface they are used in TIP tables and TIP results to name actions. (See those interfaces for more information.)

The names of atoms are case sensitive. For example, **atom1** and **atom2** are not equal, while **atom1** and **atom3** are equal.

```
atom1: ATOM = MakeAtom["Atom"L];
atom2: ATOM = MakeAtom["ATOM"L];
atom3: ATOM = Make[GetPName[atom1]];
```

The value of an atom is a function of the characters of its name and the names of the atoms that have been previously created. Atoms may not be pickled (put in a permanent representation that may be filed and recovered later) or transmitted to another system. The atom is just a convenient way to represent and manipulate the name, which is the permanent representation.

6.4 Index of Interface Items

7

AtomicProfile

7.1 Overview

The AtomicProfile interface provides a general mechanism for storing and retrieving global values, such as user name and password. Values are named by atoms and may have a type of either boolean, long integer, or string. Only one value is associated with each atom, regardless of type.

Boolean and long integer values are simple values, unlike string values, which are passed by reference. The value of strings may be gotten by calling the GetString routine, in which case they must be returned to the implementation using DoneWithString. They may be gotten by using a callback procedure in EnumerateString.

7.2 Interface Items

7.2.1 Boolean Values

```
GetBOOLEAN: PROCEDURE [atom: Atom.ATOM] RETURNS [BOOLEAN];
```

GetBOOLEAN returns the boolean value associated with atom. If no boolean value is associated with atom, GetBOOLEAN returns FALSE.

```
SetBOOLEAN: PROCEDURE [atom: Atom.ATOM, boolean: BOOLEAN];
```

SetBOOLEAN associates the boolean value boolean with atom. If atom previously had another value associated with it, that value is replaced. The event AtomicProfileChange is notified, with event data being a long pointer to atom.

7.2.2 Integer Values

```
GetLONGINTEGER: PROCEDURE [atom: Atom.ATOM] RETURNS [LONG INTEGER];
```

GetLONGINTEGER returns the long integer value associated with atom. If no long integer value is associated with atom, GetLONGINTEGER returns 0.

SetLONGINTEGER: PROCEDURE [atom: Atom.ATOM, int: LONG INTEGER];

SetLONGINTEGER associates the long integer value **int** with **atom**. If **atom** previously had another value associated with it, that value is replaced. The event **AtomicProfileChange** is notified, with event data being a long pointer to **atom**.

7.2.3 String Values

GetString: PROCEDURE [atom: Atom.ATOM] RETURNS [XString.Reader];

GetString returns the string value associated with **atom**. The string is reference-counted, and the client must return the string by calling **DoneWithString**. If there is no string value associated with **atom**, **GetString** returns NIL.

DoneWithString: PROCEDURE [string: XString.Reader];

A reader obtained by using **GetString** must be returned via **DoneWithString** so that the implementation's use-count will be correct. Failure to do so results in a storage leak if the value of the atom is replaced (see the example below).

```
EnumerateString: PROCEDURE [
   atom: Atom.ATOM, proc: PROCEDURE [XString.Reader]];
```

EnumerateString provides an alternate method of examining the string value of an atom. If **atom** has a string value, **proc** is called with the string value. **proc** is called from within the monitor of the implementation. The reader is valid for the duration of the callback, but **proc** must not call any of the operations in the implementation. If **atom** has no string value, **proc** is not called.

```
SetString: PROCEDURE [atom: Atom.ATOM, string: XString.Reader,
   immutable: BOOLEAN ← FALSE ];
```

SetString associates the string value **string** with **atom**. If **atom** previously had another value associated with it, that value is replaced. If **immutable** is FALSE, **SetString** copies **string**'s body and byte sequence; otherwise, it only copies the reader body. The client must not deallocate the byte sequence in this case. The event **AtomicProfileChange** is notified, with event data being a long pointer to **atom**.

7.3 Usage/Examples

AtomicProfile provides a general mechanism for storing and retrieving values. Actual use by a client depends on knowing the names and expected types of values. ViewPoint defines some basic values, such as user name and password. Other systems may define other values.

In the following example, a client keeps track of the user name, which depends on the **AtomicProfileChange** event. **UserNameChanged** is called when any **AtomicProfile** value is changed. By examining the event data of the agent procedure, the example can act on changes to the user name.

```
atomicProfileChange: Atom.ATOM = Atom.MakeAtom["AtomicProfileChange"L];
fullUserName: Atom.ATOM = Atom.MakeAtom["FullUserName"L];
debugging: Atom.ATOM = Atom.MakeAtom["Debugging"L];

UserNameChanged: Event.AgentProc = {
   atomChanged: LONG POINTER TO Atom.ATOM = eventData;
   IF atomChanged ↑ = fullUserName THEN {
   name: XString.Reader = GetString[fullUserName];
   << do processing of new name >>
   IF GetBOOLEAN[debugging] THEN {<< do debugging only code>>};
   DoneWithString[name]}};

Event.AddDependency[
   agent: UserNameChanged, myData: NIL, event: atomicProfileChange];
```

7.4 Index of Interface Items

8

Attention

8.1 Overview

The **Attention** interface provides a means for displaying messages to the user. It implements a single window into which messages are displayed. In addition to displaying messages, the **Attention** window has a menu to which clients can add system-wide commands.

There are three types of messages: simple messages, sticky messages, and confirmed messages. Simple messages have no special semantics. Sticky messages are redisplayed when a non-sticky message is cleared. **Attention** keeps track of one sticky message. Confirmed messages ask for confirmation by the user.

Attention allows messages to be logically appended. Each of the posting operations, **Post**, **PostSticky**, and **PostAndConfirm**, contains a boolean parameter **clear**. If **clear** is TRUE, the window is cleared before the message is displayed. If not, the message is appended to the currently displayed message. This allows the client to use **Attention** to construct complex messages.

Note that **Attention** works in concert with **BackgroundProcess**. If **Attention** is called from the Notifier process, the message is posted immediately in the attention window. If **Attention** is called from a non-Notifier process that has registered itself with the background manager by calling **BackgroundProcess.ManageMe**, then the background manager intercepts these messages and allows the user to see them later upon request (see **BackgroundProcess** for more details). *This means that **Attention** can be called from any process at any time without worry.* Fine point: In ViewPoint 1.0, there was no background manager and the following restriction applied: The **Attention** interface could only be called from the Notifier process.

To facilitate message construction, an **XFormat.Handle** is provided whose format procedure will post a simple message without clearing the window. See the example below and the **XFormat** chapter for more information.

The **Attention** window has a global system menu. Operations are provided so that clients may add menu items to this menu, remove items from the menu, or swap items in the menu.

8.2 Interface Items

8.2.1 Simple Messages

```
Post: PROCEDURE [s: XString.Reader, clear: BOOLEAN ← TRUE, beep: BOOLEAN ← FALSE,
   blink: BOOLEAN ← FALSE];
```

Post displays the message **s** in the **Attention** window. If **clear** is **TRUE**, it clears the **Attention** window before displaying **s**; otherwise, it displays **s** after whatever text is currently showing. **Attention** makes its own copy of the reader body and bytes of **s**. **beep** and **blink** stipulate that the corresponding feedback be presented to the user.

```
Clear: PROCEDURE;
```

Clear clears the **Attention** window of any simple message. If a simple message is being displayed and there is a current sticky message, the sticky message is displayed. **Clear** has no effect if a sticky message is being displayed.

```
formatHandle: XFormat.Handle;
```

formatHandle is an **XFormat.Handle** provided by the **Attention** window that clients can use to post simple messages. Its format procedure logically calls **Post** with clear being **FALSE**. (See below for an example.)

8.2.2 Sticky Messages

Sticky messages are redisplayed when a non-sticky message is cleared. **Attention** keeps track of one sticky message.

```
PostSticky: PROCEDURE [s: XString.Reader, clear: BOOLEAN ← TRUE,
   beep: BOOLEAN ← FALSE, blink: BOOLEAN ← FALSE];
```

PostSticky appends **s** to, or replaces, the current sticky message and then displays the new message in the window. Its operation is: (1) if the window has a simple message or **clear**, then clear the window; (2) if the window is clear, then clear the current sticky message; (3) append **s** to the current sticky message; and (4) display the new current sticky message. **Attention** makes its own copy of the reader body and bytes of **s**. **beep** and **blink** are the same as in **Post** above.

```
ClearSticky: PROCEDURE;
```

ClearSticky clears any current sticky message. If a sticky message is being displayed, the window is cleared. **ClearSticky** has no effect if there is no sticky message.

8.2.3 Confirmation Messages

```
PostAndConfirm: PROCEDURE [
   s: XString.Reader, clear: BOOLEAN ← TRUE, confirmChoices: ConfirmChoices ← [NIL, NIL],
   timeout: Process.Ticks ← dontTimeout,
  beep: BOOLEAN ← FALSE, blink: BOOLEAN ← FALSE]
  RETURNS [confirmed, timedOut: BOOLEAN];
```

```
ConfirmChoices: TYPE = RECORD [yes, no: XString.Reader];
```

```
dontTimeout: Process.Ticks = 0;
```

PostAndConfirm acts like **Post** in displaying the message **s** but waits for confirmation by the user. The **confirmChoices** messages are displayed, and the user should select one of the choices with the mouse. If the user selects yes, **confirmed** is returned TRUE; if no is selected or the STOP key is depressed, **confirmed** is returned FALSE. If **confirmChoices.yes # NIL** and **confrmChoices.no = NIL**, then only **confirmChoices.yes** is posted and **confirmChoices.no** is ignored. This is useful for posting a message that the user must see, but for which the user gets no choice, such as "Unable to communicate with the printer: CONTINUE". **PostAndConfirm** absorbs all user input except the STOP key and mouse actions over the yes and no messages. The client may specify a **timeout** value, which causes **PostAndConfirm** to return **confirmed** FALSE and **timedOut** TRUE if the user does not act within **timeout** ticks. The default value **dontTimeout** disables this timeout feature. **Attention** makes its own copy of the reader body and bytes of **s**.

8.2.4 System Menu

```
AddMenuItem: PROCEDURE [item: MenuData.ItemHandle];
```

AddMenuItem adds **item** to the global system menu.

```
RemoveMenuItem: PROCEDURE [item: MenuData.ItemHandle];
```

RemoveMenuItem removes **item** from the global system menu. There is no effect if **item** is not in the menu.

```
SwapMenuItem: PROCEDURE [old, new: MenuData.ItemHandle];
```

SwapMenuItem swaps **new** for **old** in the global system menu. **SwapMenuItem[old: NIL, new: item]** is equivalent to **AddMenuItem[item: item]** and **SwapMenuItem[old: item, new: NIL]** is equivalent to **RemoveMenuItem[item: item]**.

8.3 Usage/Examples

The following example has a client displaying the name and size of a file. The example uses the **NSFile** interface to access the file and get the name and size attributes. See the *Services Programmer's Guide* (610E00180): *Filing Programmer's Manual* for documentation on the **NSFile** interface.

```
PostNameAndSize: PROCEDURE [file: NSFile.Handle ] = {
  nameSelections: NSFile.Selections = [interpreted: [name: TRUE]];
  attributes: NSFile.AttributesRecord;
  rb: XString.ReaderBody ← Message[theFile];
  Attention.Post[s: @rb, clear: TRUE]; -- start a new message
  NSFile.GetAttributes [file, nameSelections, @attributes];
  XFormat.NSString[Attention.formatHandle, attributes.name];
  NSFile.ClearAttributes [@attributes];
XFormat.ReaderBody[h: Attention.formatHandle, rb: Message[contains]];
XFormat.Decimal[h: Attention.formatHandle, n: NSFile.GetSizeInBytes[file]];
rb ← Message[bytes];
Attention.Post[s: @rb]}; -- clear defaults to FALSE

Message: PROCEDURE [key: {theFile, contains, bytes}] RETURNS [XString.ReaderBody] = {
   ...};
```

An example of the resulting message displayed in the **Attention** window is

```
The file Foo contains 53324 bytes
```

The example intermixes use of the format handle and use of the **Post** procedure. A client could clear first, using the **Clear** procedure, and then display the message by just using the format handle. **Note:** In a multilingual environment constructing a sentence from pieces like this is not recommended because the grammar of other languages could cause this sentence to be rather confusing.

8.4 Index of Interface Items

9

BackgroundProcess

9.1 Overview

BackgroundProcess provides basic user feedback and control facilities to clients that want to run in a process other than the the Notifier process (see §3.4.1). Once registered with BackgroundProcess, the client process can use Attention to post messages, and check to see if the process has been aborted by the user. The user can look at the messages posted by the process, and abort the process. Fine Point: The implementation of BackgroundProcess is a plugin, so the user interface is up to a particular background manager. See ViewPoint friends level documentation for details on how to build a background manager.

9.2 Interface Items

```
ManageMe: ManageProc;

ManageProc: TYPE = PROCEDURE [
   name: XString.Reader,
   callBackProc: CallBackProc,
   window: Window.Handle ← NIL,
   icon: Containee.DataHandle ← NIL,
   context: LONG POINTER ← NIL,
   abortable: BOOLEAN ← FALSE
   ]
   RETURNS [finalStatus: FinalStatus];

CallBackProc: TYPE = PROCEDURE [context: LONG POINTER]
   RETURNS [finalStatus: FinalStatus];

FinalStatus: TYPE = MACHINE DEPENDENT{
   importantFailure(0), failure, quietSuccess, success, aborted, firstFree, last(15)};

quietIfNoUnreadMsg:FinalStatus = firstFree;
quietIfNoUnreadImportantMsg:FinalStatus = SUCC[firstFree];
```

A client process that wishes to be managed calls ManageMe. The client should already be in the process that it wishes to have managed; if the client starts in the Notifier, the client should do a FORK and call ManageMe from the forked process. name is a string that may be used by the background manager to identify the process to the user; the bytes in name are copied by the background manager. After **ManageMe** is called, the background manager will call **callBackProc** with **context** to give control back to the client process. If the process is prepared to catch ABORTED, then **abortable** should be TRUE. If the process is *not* prepared to catch ABORTED, then **abortable** should be FALSE. (see §9.3.2). **window** and **icon** may be provided for use by the background manager; if the process is tied to a particular window or icon, the background manager may use these to allow the user to manipulate the process via the window or icon. When the client process is completed or aborted, it should return from **callBackProc** with a **finalStatus** indicating the outcome of the process. **importantFailure** indicates that the user should be warned that the process terminated in a way that might need the user's attention. **failure** indicates that the process failed in some way but that we don't need to inform the user in any special way. **quietSuccess** indicates that the process should go away without any final notice to the user. **success** indicates that process succeeded and that a final status message may be posted. **aborted** indicates that the process was aborted by the user. **quietIfNoUnreadMsg** tells the background manager that if there are no unseen messages for this process, terminate as it would with a status of **quietSuccess**. If messages remain to be read, the termination is treated as a status of **success**. **quietIfNoUnreadImportantMsg** tells the background manager to terminate as a **quietSuccess** if there are no flagged messages (see **FlagImportantMsg**). **Fine point:** **quietIfNoUnreadMsg** and **quietIfNoUnreadImportantMsg** are defined in **BackgroundProcessExtra**.

```
UserAbort: PROCEDURE [process: PROCESS ← nullProcess] RETURNS [BOOLEAN];

ResetUserAbort: PROCEDURE [process: PROCESS ← nullProcess];

AbortProc: TYPE = PROCEDURE [context: LONG POINTER];
SetAbortProc: PROCEDURE [abortProc: AbortProc, process: PROCESS ←NIL];

nullProcess: PROCESS = LOOPHOLE[0];
```

Clients of the background manager have a choice about how they are notified when the user tries to abort a background task. See §9.3.2 for more details about how these choices interact.

UserAbort returns TRUE if the user has requested that the process be aborted. **ResetUserAbort** clears any pending abort; if the user has requested an abort, **UserAbort** will return TRUE until **ResetUserAbort** is called or the process terminates..The client can also call **SetAbortProc** to specify an **AbortProc** that will be called when the user tries to abort a process. The **AbortProc** will be passed the context pointer that was passed into **ManageMe**; therefore, **SetAbortProc** can only be called after the client has called **ManageMe**. .**Fine point:** **AbortProc** and **SetAbortProc** are defined in **BackgroundProcessExtra**.

For **SetAbortProc**, **UserAbort**, and **ResetUserAbort**, **process** is defaulted to **nullProcess**. All three procedures assume that the current process is the process that called **ManageMe**. **process** should only be used if the process calling **UserAbort**, **ResetUserAbort**, or **SetAbortProc** is different from the process that called **ManageMe**. **Fine point:** **nullProcess** is equivalent to **NIL**. **nullProcess** will be removed from future versions of the interface.

```
FlagImportantMsg: PROCEDURE [message, comment: XString.Reader ←NIL, PROCESS ←NIL];
```

This procedure lets the background manager know that there is a message that the user should see. If the client process terminates with a status of **quietIfNoUnreadErrorMsg** before the message is read, the background manager will make sure that task is still available to allow the user to read the message. The background manager may also use this call to may some kind of visual notification to the user that an important message is available. The background manager supplied by the Basic Workstation displays a property sheet and waits for the user to respond. If **message** is NIL, the message posted is the current message available in the background manager. If **message** is non-NIL, that **message** will be posted instead. If **comment** is non-NIL, an extra string will be posted at the end of the sheet which may be used to indicate error recovery. If **comment** is NIL, this part of the psheet is not visible. **FlagImportantMsg** is synchronous: it will not return until the user bugs done on the property sheet. Since the background manager is a plugin, other background mangers may behave differently. **Fine point:** This procedure is currently exported by **BackgroundProcessExtra.**

```
GetName: PROCEDURE [process: PROCESS ← NIL] RETURNS [name: XString.ReaderBody];
SetName: PROCEDURE [newName: XString.Reader, process: PROCESS ← NIL];
```

These procedures allow the client to manipulate the name of the task. The name is originally set by the **name** parameter to **ManageMe**; these procedures allow the client to change that name. The name is typically used by the background manger to label the task for the user. **SetName** copies the bytes in **newName**. The bytes from **GetName** belong to the background manager and should be copied if the client wishes to use them.

```
Mode: TYPE = {foreground, background};
mode: READONLY Mode;
```

mode indicates whether applications should FORK background processes or not. Before FORKing a background process, applications should check **mode** and if it is foreground, do not do the FORK, but rather do the operation in the foreground process. This is primarily used during Cusp programs to synchronize each Cusp statement.

```
backgroundCount: READONLY CARDINAL;
```

backgroundCount is the current number of background activities registered with the background manager.

```
cuspIsRunning: BOOLEAN;
```

Cusp sets this to TRUE during execution of a Cusp program. Applications can interpret this in whatever way is appropriate, for example by not posting option sheets.

9.3 Usage/Examples

9.3.1 Posting Messages

Once a client process has called **ManageMe**, it can freely post messages using **Attention**. Fine point: the exact method the messages will be displayed is up to the background manager. Also, only the client process that originally called **ManageMe** can call **Attention** directly. If a background process has any associated subprocesses that need to use **Attention** to post messages, it must use a friends level **Attention** interface to associate the subprocess with the client's main background process.

9.3.2 Aborting processes

A client of the background manager can be notified when the user tries to abort a background task. There are three ways that the client can be notified.

If the client calls **ManageMe** with **abortable** = TRUE, the background manager will call Process.**Abort** on the process that called **ManageMe**. That process should be prepared to catch ERROR ABORTED.

The client may also call **SetAbortProc** with a procedure that will be called if the user tries to abort. This procedure will be called only if **ManageMe** was called with **abortable** = FALSE; if **abortable** = TRUE, the manager will call Process.**Abort** instead of calling the **AbortProc**.

Finally, the client may also call **UserAbort** at any time. If the client does not enable the use of Process.**Abort** or set an **AbortProc**, it is the client's responsibility to periodically call **UserAbort** to see if the user has tried to abort the process. If the client does not check **UserAbort**, user attempts at stopping the process will have no effect. The client may call **UserAbort** from inside an **AbortProc**.

9.3.3 Example

This example program fragment illustrates the structure of a typical use of **BackgroundProcess**. In this example, a **MenuProc** is provided that can be called from the attention window. The **MenuProc** immediately forks a process, which reduces its priority and then calls **BackgroundProcess**. The example program posts four messages, pausing between each, and checking **UserAbort** on each pass.

```
backgroundName: XString.ReaderBody ←XString.FromSTRING["Background Post"L];
abortedString: XString.ReaderBody ←XString.FromSTRING["Process canceled ..."];

Init: PROCEDURE = {
  Attention.AddMenuItem [
    MenuData.CreateItem [
    zone: z,          -- some private zone
    name: @backgroundName,
    proc: BackgroundProcessPost] ];

BackgroundProcessPost: MenuData.MenuProc = {
  Process.Detach [FORK DoBackgroundProcessPost[s: @backgroundName]]};

DoBackgroundProcessPost: PROCEDURE [s: XString.Reader] = {
  DoIt: BackgroundProcess.CallBackProc = {
    FOR i: CARDINAL IN [1..4] DO
      IF BackgroundProcess.UserAbort[] THEN {
        Attention.Post[@abortedString];
        RETURN[aborted]};
      Attention.Post [s: s];
      Attention.formatHandle.Blanks[2];
      Attention.formatHandle.Decimal[i];
      Process.Pause [Process.SecondsToTicks[10]];
      ENDLOOP;
    RETURN [success]};

  Process.SetPriority[Process.priorityBackground];
  [] ← BackgroundProcess.ManageMe [name: @backgroundName, callBackProc: DoIt]};
```

9.4 Index of Interface Items

10

BlackKeys

10.1 Overview

The BlackKeys interface changes the interpretation of the main (central) section of the physical keyboard. It includes the data structures that define a keyboard record as well as the procedures used to manipulate the keyboard stack.

The average client uses only the *data structures* that the BlackKeys interface provides. The procedures are reserved for a keyboard manager interested in interfacing between the user and the blackkeys stack of keyboards.

10.2 Interface Items

10.2.1 Keyboard Data Structures

The BlackKeys data structures provide the framework for client-defined keys in the main (central) section of the physical keyboard. This includes interface to a keyboard picture whose keytops may be selected with the mouse to simulate pressing the physical key on the keyboard.

```
Keyboard: TYPE = LONG POINTER TO KeyboardObject ← NIL;

KeyboardObject: TYPE = RECORD [
   table: TIP.Table ← NIL,
   charTranslator: TIP.CharTranslator ← [proc: NIL, data: NIL],
   pictureProc: PictureProc ← NIL,
   label: XString.ReaderBody ← XString.nullReaderBody,
   clientData: LONG POINTER ← NIL];
```

KeyboardObject is the keyboard interpretation data structure. The client may provide its own TIP.Table or default it to NIL, in which case the NormalKeyboard.TIP table is used. (See Appendix A for productions returned by NormalKeyboard.TIP). A TIP.CharTranslator may be provided to handle CHAR and BUFFEREDCHAR productions from a TIP.Table. A PictureProc may be provided to be called when installing or removing this keyboard. Absence of such a procedure assumes no picture is associated with this keyboard. label is the string that appears in the SoftKeys window when the KEYBOARD key is pressed down. Pressing (or selecting) the key marked label invokes this keyboard. clientData is provided to associate any other information the client might need to keep with the keyboard.

```
PictureProc: TYPE = PROCEDURE [
   keyboard: Keyboard,
   action: PictureAction]
   RETURNS [
   picture: Picture ← nullPicture,
   geometry: GeometryTable ←NIL];
```

PictureProc is a client-provided procedure that is called by a keyboard window application when the client's keyboard is being installed (action = acquire) or removed (action = release) from the top of the blackkeys stack of active keyboards. The client may use this opportunity to map or unmap the picture and geometry table that the keyboard window application uses.

```
PictureAction: TYPE = {acquire, release};
```

acquire = client's keyboard is being installed at the top of the keyboard stack (becoming the current keyboard).

release = client's keyboard is being removed from the top of the keyboard stack.

```
PictureType: TYPE = {bitmap, text};

Picture: TYPE = RECORD [
  variant: SELECT type: PictureType FROM
   bitmap = > [bitmap: LONG POINTER],
   text = > [text: XString.Reader]
   ENDCASE];
```

The variant of the record, Picture, allows the client to present its keyboard window in either bitmap or textual form. (See the KeyboardWindow interface for a discussion of the structure behind a keyboard bitmap.) text is pointed to by an XString.Reader. The text is not copied.

```
nullPicture: bitmap BlackKeys.Picture = [bitmap[NIL]];
```

The variable nullPicture represents a null entry to the keyboard window.

```
GeometryTable: TYPE = LONG POINTER;
```

A geometry table allows access to the data structure. (See the KeyboardWindow interface chapter for discussion of the structure of a geometry table.)

10.2.2 Getting a Handle to the Current Keyboard

```
BlackKeysChange: Event.EventType; -- ATOM defined as "BlackKeysChange"
```

Changing the keyboard at the top of the blackkeys stack of keyboards results in the notification BlackKeysChange through the Event mechanism. The eventData supplied by the Event.Notify is the current keyboard handle.

GetCurrentKeyboard: PROCEDURE RETURNS [current: Keyboard];

GetCurrentKeyboard returns the current keyboard from the top of the blackkeys stack.

10.2.3 Procedures

The following procedures are NOT expected to be used by Applications programmers. Instead see KeyboardKey.SetKeyboard.

Push: PROCEDURE [keyboard: Keyboard];

The Push procedure installs a black key interpretation at the top of the blackkeys stack of keyboards. The TIP.Table and/or TIP.CharTranslator are registered with TIP and the event BlackKeysChange is broadcast.

Remove: PROCEDURE [keyboard: Keyboard];

The Remove procedure removes the keyboard from the stack of active keyboards and resets the TIP.Table and TIP.CharTranslator as applicable. The event BlackKeysChange is broadcast if keyboard is on the top of the blackkeys stack.

May raise the ERROR BlackKeys.InvalidHandle.

Swap: PROCEDURE [old:Keyboard, new:Keyboard];

The Swap procedure is designed to change black keys' interpretations without returning to some previous or other default value in between. It is essentially the equivalent of a Remove followed by a Push. The event BlackKeysChange is broadcast if the keyboard being removed was on top of the stack.

May raise the ERROR BlackKeys.InvalidHandle.

10.2.4 Errors

InvalidHandle: ERROR;

This error is raised if the keyboard passed to Remove or Swap (old) is not in the set of active BlackKeys keyboards.

10.3 Usage/Examples

10.3.1 Defining a Keyboard Record

```
DefineKeyboard: PROCEDURE =
BEGIN
 nameString: XString.ReaderBody ← XString.FromSTRING["Swahili"L]
 swahiliKeyboardRecord: BlackKeys.KeyboardObject ←[
  table: NIL,
  charTranslator: [MakeChar, NIL],
  pictureProc: MapBitmapFile,
  label: XString.CopyToNewReaderBody[@nameString, Heap.systemZone]];
 --Save the pointer to the record somewhere for future use --
END; --DefineKeyboard --

MapBitmapFile: BlackKeys.PictureProc =
BEGIN
 pixPtr: BlackKeys.Picture.bitmap ←BlackKeys.nullPicture;
 SELECT action FROM
  acquire =>
{--Do the right thing to map the bitmap. Uses the default geometry table. --
   RETURN[pixPtr, KeyboardWindow.defaultGeometry] };
  release => {--Do the right thing to unmap the bitmap --
   RETURN[BlackKeys.nullPicture, NIL] }
END; -- MapBitmapFile

MakeChar: TIP.KeyToCharProc =
BEGIN
 --Map bufferedChar to desired XString.Character --
END; -- MakeChar
```

10.4 Index of Interface Items

11

BodyWindowParent

11.1 Overview

BodyWindowParent provides a facility for creating body windows in a subwindow. The client may provide several procs for dealing with scrolling multiple body windows within the parent viewing region. For a comprehensive view of all the subwindow interfaces and their intended use, see the Subwindow Overview chapter.

11.1.1 Body windows Discussion

A bodyParent window has an interior window that is a child of the bodyParent and is exactly the size of the available window space in the bodyParent (that is, the bodyParent minus its scrollbars). The interior window may have child windows created by the client. These children of the interior window are called *body windows*. The client may create an arbitrary number of body windows and arrange them in an arbitrary fashion. Note: Since the body windows are children of the interior window, they are clipped by the interior window. A client could, for example, create a body window that is very much taller than the interior window and accomplish scrolling by simply sliding the body window around inside the interior window. (This is actually what the default scrolling does; for more detail, see this Chapters section on scrolling).

Body windows are created by calling BodyWindowParent.CreateBody. This returns a Window.Handle. The client can create an arbitrary number of body windows. Each body window is a child of the bodyParent's interior window. The body windows may overlap or not. They can be in any arrangement the client finds useful. Some common arrangements of body windows are as follows:

- One very long body window.
  This is easy to scroll by simply sliding the body window, which is what the default scrolling does.

- One body window with BodyWindowJustFits = TRUE.
  This is one way to display an infinite amount of data, such as a Tajo-like editor. The client must keep track of what is currently in the window, use adjust procs, do scrolling, and so forth. This is difficult to implement.

- Several body windows about the size of the interior, adjacent, non-overlapping. This is another way to display an infinite amount of data. The client lets BodyWindowParent do default scrolling, which slides the body windows up or down and then calls the client to supply more body windows when it runs out. The client might put one page of text into each body window, supplying pages to BodyWindowParent scrolling as needed.

- Several body windows smaller than the interior, adjacent, non-overlapping.

Note: Body windows can themselves have child windows, and so on. A client might implement frames in a document editor by making each frame a child of a body window.

11.2 Interface Items

11.2.1 BodyParent window

```
Create: PROCEDURE [
    parent: Window.Handle,
    verticalScrollbar: BOOLEAN ← TRUE,
    horizontalScrollbar: BOOLEAN ← TRUE,
    adjustProc: AdjustableWindow.AdjustProc ← NIL,
    garbageCollectBodiesProc: GarbageCollectBodiesProc ← NIL,
    moreScrollProc: MoreScrollProc ← NIL,
    scrollbarInfoProc: Scrollbar.ScrollbarInfoProc ← NIL,
    thumbFeedbackProc: Scrollbar.ThumbFeedbackProc ← NIL,
    thumbScrollProc: Scrollbar.ThumbScrollProc ← DefaultThumbScroll,
    zone: UNCOUNTED ZONE];
```

Takes a window and makes it a body parent window. An interior window is created (child of the body parent) that becomes the viewing region for future body windows. verticalScrollbar and horizontalScrollbar are attached as instructed by the client provided BOOLEANS. The scrollbarInfoProc will be called when the user thumb scrolls and then the appropriate feedback (like the ViewPoint diamond or the tajo bar) will be painted in the thumbing region of the scrollbar. Fine Point: The client should call GetInteriorDims when deterimining the offset and portion. moreScrollProc and garbageCollectBodiesProc are described in §2.3.

```
IsIt: PROCEDURE ]
    parent: Window.Handle
    RETURNS [BOOLEAN];
```

Determines if parent is indeed a bodyParent created by calling BodyWindowParent.Create. Returns TRUE if it is, FALSE if not.

```
Destroy: PROCEDURE [parent: Window.Handle];
```

Destroy destroys the parent window, its descendents and any associated data.

```
Adjust: AdjustableWindow.AdjustProc;
```

Adjust should be called when the body parent is being resized (before and after the SlideAndSize).

```
SetAdjustProc: PROCEDURE [
    parent: Window.Handle,
    new: AdjustableWindow.AdjustProc]
    RETURNS [old: AdjustableWindow.AdjustProc];
```

new will be called whenever **parent** is changing size.

11.2.2 Body windows

```
CreateBody: PROCEDURE [
    parent: Window.Handle ← NIL,
    box: Window.Box ← [[0,0], [0,INTEGER.LAST]],
    displayProc: Window.DisplayProc,
    notifyProc: TIP.NotifyProc,
    clearingRequired:BOOLEAN ← TRUE,
    windowPane: BOOLEAN ← FALSE,
    under, cookie, color: BOOLEAN ← FALSE ]
    RETURNS [body: Window.Handle];
```

Creates a **body** window that is a descendent of **parent**. If **parent** is NIL, body will be an orphaned window that can be installed at a later date. (See **InstallBody** below.) If **box**.dims.w = 0 THEN box.dims.w ← size of parent's interior. If **box**.dims.h = 0 THEN box.dims.h ← size of parent's interior. **notifyProc** will be attached to **body** and TIPStar.NormalTable[]. **clearingRequired**, **windowPane**, **under**, **cookie**, and **color** are described in the **Window** chapter. If a body is created within a visible parent, the client must call Window.ValidateTree[body] to effect the change on the screen.

```
InstallBody: PROCEDURE[body: Window.Handle, parent: Window.Handle];
```

Installs the **body** window in the tree of **parent** as the eldest sibling.

```
DestallBody: PROCEDURE[body: Window.Handle];
```

Removes the **body** from its parent's window tree.

```
DestroyBody: PROCEDURE [body: Window.Handle];
```

Destroys the **body** window and associated data.

11.2.3 Scrolling

Only part of an object is usually visible to the user at any one moment in the interior of a bodyParent. The user can request to see more of the object by scrolling the contents up or down inside the bodyParent. The user can perform three kinds of scrolling by using the scrollbars. (1) He can move the contents a little at a time by pointing at the arrows (up, down, left, right) in the scrollbars. (2) He can move the contents a page or screenful at a time by pointing at the plus (+) and minus (-) signs. (3)He can jump to any arbitrary place within the entire extent of the object being viewed by pointing in the blank part of the vertical scrollbar (this latter operation is called *thumbing*).

BodyWindowParent provides various levels of support to a client for performing these scrolling operations. The client can allow **BodyWindowParent** to do all the scrolling functions, the client can do some of them and leave the rest to **BodyWindowParent**, or the client can do all scrolling operations. Much of this decision will be based on how the client chooses to arrange body windows within the bodyWindowParent (see the section on body windows above and more discussion below). First, we will describe the various types of scrolling and scrolling procedures that a client can supply; then we will describe the default scrolling procedures provided by **BodyWindowParent**.

In the simplest (for the client) case, one body window contains the entire extent of the object being viewed. **BodyWindowParent** can handle all scrolling in this case. The client simply does nothing. When the user points at an arrow, **BodyWindowParent** moves the body window a small amount. When the user point at plus or minus, **BodyWindowParent** moves the body window by one interior window's height. When the user thumbs, **BodyWindowParent** will move the body window to an appropriate place based on its overall height.

In a slightly more complex case, body windows are butted up against one another. When the user points at an arrow, **BodyWindowParent** moves all the body windows a small amount. When the user point at plus or minus, **BodyWindowParent** moves all the body windows by one interior window's height. When the user thumbs, **BodyWindowParent** moves all the body windows to an appropriate place based on the combined overall height of the body windows. However, in this case the client often does not have the entire extent of the object displayed in these body windows but rather wants to tack new body windows on each end as these body windows are scrolled off. The client can do this by providing a **MoreScrollProc** for the shell. **BodyWindowParent** calls the client's **MoreScrollProc** whenever it runs out of body windows.

In the most complex case, the client has a single body window that "just fits" (see **SetBodyWindowJustFits** in the section on body windows), and only part of the entire object is displayed at any one time. The client must provide *all* the scrolling functions for this case. This means providing a **Scrollbar.SingleScrollProc** (to handle the user's pointing at the arrows, plus, and minus) and a **Scrollbar.ThumbScrollProc** (to handle the user's thumbing). See the Chapter on Scrollbar for further information on setting the **SingleScrollProc**.

Of course, the client may provide its own scrolling procedures for any of the above cases, even the simple one, to override the type of scrolling that **BodyWindowParent** provides. But at some point it would be wiser to register a new subwindow type instead. If BodyWindowParent type AttachScrollbars and Adjustment are desireable then the client should use the Standard subwindow procs exported by SubwindowFriends when creating their own unique type.

11.2.3.1 ScrollProcs

The following ScrollProc types are passed in by the client when creating a bodyParent window. Any or all of them may be defaulted to NIL.

```
MoreScrollProc: TYPE = PROCEDURE [
    parent: Window.Handle,
    type: Scrollbar.Type,
    flavor: MoreFlavor,
    amount: CARDINAL];
```

```
MoreFlavor: TYPE = {before, after};
```

The **MoreScrollProc** is called when we run out of body windows during scrolling. (See discussion above) **.amount** is pixels.

The client's **moreScroll** procedure is responsible for adding and deleting body windows from the bodyParent. The case being handled is that in which the client has a large number of pages to display to the user and wishes to manifest only a few. Then we need to handle the case in which system scrolling would make a non-manifest page visible, and there is no body window for it. Whenever the system is about to perform a scroll function, it checks to see if the scroll action would move the visible portion of the bodies off the end of the existent body windows. If so, it calls a non-nil client **MoreScrollProc**, indicating how much more body window may be displayed. The client may augment the collection of body windows or not. The system routines will not scroll past the end of the body windows.

```
GarbageCollectBodiesProc: TYPE = PROCEDURE [
    parent: Window.Handle,
    body: Window.Handle,
    type: Scrollbar.Type];
```

Called when **body** is no longer visible. Allows client a chance to destroy or reuse the body.

11.2.3.2 Getting and Setting ScrollProcs

The following procedures Get and Set the ScollProcs associated with **parent.**

```
GetScrollProcs: PROCEDURE [parent: Window.Handle]
    RETURNS [
    garbageCollectBodiesProc: GarbageCollectBodiesProc,
    moreScrollProc: MoreScrollProc,
    scrollbarInfoProc: Scrollbar.ScrollbarInfoProc
    thumbFeedbackProc: Scrollbar.ThumbFeedbackProc,
    thumbScrollProc: Scrollbar.ThumbScrollProc];

SetMoreScrollProc: PROCEDURE [
    parent: Window.Handle,
    new: MoreScrollProc]
    RETURNS [old: MoreScrollProc];

SetScrollbarInfoProc: PROCEDURE [
    parent: Window.Handle,
    new: Scrollbar.ScrollbarInfoProc]
    RETURNS [old: Scrollbar.ScrollbarInfoProc];

SetGarbageCollectBodiesProc: PROCEDURE [
    parent: Window.Handle,
    new: GarbageCollectBodiesProc]
    RETURNS [old: GarbageCollectBodiesProc];
```

```
SetThumbFeedbackProc: PROCEDURE [
    parent: Window.Handle,
    new: Scrollbar.ThumbFeedbackProc]
    RETURNS [old: Scrollbar.ThumbFeedbackProc];

SetThumbScrollProc: PROCEDURE [
    parent: Window.Handle,
    new: Scrollbar.ThumbScrollProc]
    RETURNS [old: Scrollbar.ThumbScrollProc];
```

11.2.3.3 Default ScrollProcs

```
DefaultSingleScroll: Scrollbar.SingleScrollProc;

DefaultScrollbarInfo: Scrollbar.ScrollbarInfoProc;

DefaultThumbScroll: Scrollbar.ThumbScrollProc;
```

Calling DefaultSingleScroll, DefaultScrollbarInfo or DefaultThumbScroll will invoke standard scrolling of window in specified flavor and amount. Can be used to set the desired Procs or called independently. The type of scrolling provided is as described in the General Discussion Simple Case in Sections 1.1 and 2.3

11.2.4 Utilities

```
GetZone: PROCEDURE [parent: Window.Handle] RETURNS[zone: UNCOUNTED ZONE];
```

Returns the zone associated with the bodyParent window.

```
ParentFromBody: PROCEDURE [body: Window.Handle]
    RETURNS [parent: Window.Handle];
```

Given a body window returns its body parent subwindow.

```
GetBody: PROCEDURE [parent: Window.Handle]
    RETURNS [body: Window.Handle];
```

Returns the first body window in parent.

```
GetScrollbar: ROCEDURE [window: Window.Handle, type: Scrollbar.Type]
    RETURNS [scrollbar: Window.Handle];
```

Given a body window or bodyParent window, returns the associated scrollbar of type.

```
GetInteriorDims: PROCEDURE [parent: Window.Handle]
    RETURNS [dims: Window.Dims];
```

Returns the dimensions of the viewing region of the body parent subwindow.

```
IsBodyWindowOutOfInterior: PROCEDURE [body: Window.Handle]
    RETURNS [BOOLEAN];
```

Returns TRUE If all of body is sticking out of the viewing region, FALSE if any part of body is within the viewing region.

```
EnumerateBodies: PROCEDURE [
    parent: Window.Handle,
    proc: BodyEnumProc]
    RETURNS [Window.Handle];

EnumerateBodiesInDecreasingY:PROCEDURE [
    parent: Window.Handle,
    proc: BodyEnumProc]
RETURNS [Window.Handle];

EnumerateBodiesInIncreasingY:PROCEDURE [
    parent: Window.Handle,
    proc: BodyEnumProc]
RETURNS [Window.Handle];
```

The EnumerateBodiesxxx procedures enumerate all the body windows in parent, calling proc for each body window until proc returns stop = TRUE. EnumerateBodies enumerates the bodies in the order in which they appear in the parent tree. EnumerateBodiesInIncreasingY enumerates the body windows in increasing order of bodyBox.place.y, and EnumerateBodiesInDecreasingY enumerates the body windows in decreasing order of place.y. Each procedure returns the last body window enumerated or NIL if all body windows were enumerated. These procedures are especially handy for clients that do their own scrolling. To minimize repainting when scrolling a set of body windows upward, it is important to move the upper ones first, and vice versa.

```
BodyEnumProc: TYPE = PROC [body: Window.Handle]
    RETURNS [stop: BOOLEAN ←FALSE];
```

stop = TRUE will terminate the enumeration. Fine Point: destalling or destroying body within the BodyEnumProc is allowed.

11.2.5 Errors

```
Error: ERROR [code: ErrorCode];

ErrorCode: TYPE = {noBodiesInParent, notABodyWindowParent};
```

11.3 Usage/Examples

```
--In impl for Body subwindows:
AttachScrollbarsToBodySW:SubwindowFriends.AttachScrollbarsProc = {
BodyWindowParent.Create[...]. }
```

11.4 Index of Interface Items

12

BusyIcon

12.1 Overview

BusyIcon provides the client with a way to make file-backed icon object "busy." An object that is busy cannot be operated on by the user. The notion of an object being busy is a user-interface level notion; clients are still responsible for obtaining any necessary locks on an object. Making an object busy insures that normal user operations (open, drop-on, props, and so forth) cannot be invoked. If an object to be made busy is visible on the desktop or in a container, the appearance of that object will change to make the object appear busy to the user. Whether or not the object is visible, the object is still marked as busy in the Containee cache so that the next time the object is visible, it will appear busy. See the Containee chapter for documentation on the Get/SetCachedBusy operations.

12.2 Interface Items

```
IsBusy: PROCEDURE [ref: NSFile.Reference, w: Window.Handle ← NIL]
  RETURNS [yes: BOOLEAN];
```

Returns TRUE if ref is busy. If non-nil, w is used as a hint for where to look: either the desktop window or a FileContainerSource-backed container window that ref might be found in. If ref is not found in w, IsBusy will search the desktop and all open FileContainerSources for ref. If ref is still not found, IsBusy will check the Containee cache to see if the object is busy (see the Containee chapter for more information.)

IsBusy will only find files in container windows backed by a FileContainerSource or a source built on FileContainerSource.

```
BusyStatus: TYPE = {succeeded, notFound, stateNotChanged, notAllowed};
```

```
MakeBusy: PROCEDURE [ref: NSFile.Reference, w: Window.Handle ← NIL]
  RETURNS [result: BusyStatus];
```

```
MakeUnbusy: PROCEDURE [ref: NSFile.Reference, w: Window.Handle ← NIL]
  RETURNS [result: BusyStatus];
```

MakeBusy looks for ref on the desktop or in an open container window backed by a FileContainerSource. w is a hint that ref is in that window; w may be the desktop window

or a container window. If **w** is **NIL**, **MakeBusy** and **MakeUnbusy** look in all open **FileContainerSource**-backed container windows and on the desktop. Both operations return **result** to indicate the outcome of the operation. If **ref** is part of the selection, the selection is cleared (but see **notAllowed** status below.)
succeeded indicates that the icon for **ref** was found and the icon picture was changed.
notFound means that an object for **ref** was not found. The icon picture was not changed, but the Containee cache was updated.
stateNotChanged means that **ref** was found but was already in the desired state.
notAllowed is returned if the operation is not allowed at the moment. This can occur if the user calls **MakeBusy** from the background but the object is part of the user's selection. **MakeBusy** will not make the object busy, but instead will return a status of **notAllowed**.

IsBusy will only find files in container windows backed by a **FileContainerSource** or a source built on **FileContainerSource**. Files in other types of container sources will not have their picture changed by these procedures.

12.2 Usage

Clients are responsible for making sure an object is properly locked before making the object busy. If an object is just made busy without acquiring any other locks, there is nothing to stop some other program that has a reference to a file from operating on that file.

A typical way to use this interface may to do some background operation on an icon. The flow would look something like this.

- The user selects an icon and invokes a menu command.
- In the menu command (and thus in the foreground) the command does a Selection.**Convert** to get a reference to the file.
- The menu command opens the file and get an NSFile lock on
- The command calls **MakeBusy** to make sure the user cannot operate on the file.
- The command FORKs some operation.
- When the operation is done, the client would unlock the file and closes the file handle
- To finish up, the client calls **MakeUnbusy** to return the icon the the user's control.

In this example, the client does not need to worry about whether the icon was on the desktop or inside a folder. **BusyIcon** took care of making the icon busy.

12.3 Index of Interface Items

13

BWSAttributeTypes

13.1 Overview

BWSAttributeTypes defines the NSFile.ExtendedAttributeTypes that are used by ViewPoint as well as the first NSFile.ExtendedAttributeType available for client use.

The only extended attributes defined here are the ones that can be attached to any file, such as mailing and filing application attributes. Attributes that are unique to a particular application's files should be defined privately within that application. Several applications can use the same extended attributes because application A should never be reading the attributes from application B's files and vice versa. Fine point: Several application-specific attribute types are included in this interface for compatibility.

The extended attributes that can be attached to any file, leaving a few spare ones for future use. are defined here. Also defined are the first available "application attribute" (firstAvailableApplicationType). **Caution:** An application should not use an extended attribute smaller than this one, nor should an application use an extended attribute larger than lastBWSType.

13.2 Interface Items

13.2.1 Available Application Types

firstAvailableApplicationType: NSFile.ExtendedAttributeType = ...;

lastBWSType: NSFile.ExtendedAttributeType = ...;

Applications should only use the types in the range [firstAvailableApplicationType .. lastBWSType]. firstAvailableApplicationType is the first extended attribute type that applications can use to store application-specific attributes. **Caution:** An application should not use an extended attribute smaller than firstAvailableApplicationType. lastBWSType is the last extended attribute type that applications can use to store application-specific attributes. **Caution:** An application should not use an extended attribute larger than lastBWSType.

If a Viewpoint client needs more attributes than the number in this range, see the NSFiling group to obtain a range specific to that client.

13.2.2 Viewpoint Types

Consult the Mesa interface for the exact assignment of ViewPoint-specific types.

13.3 Index of Interface Items

14

BWSFileTypes

14.1 Overview

BWSFileTypes defines several **NSFile.Types** used by ViewPoint. Applications should not use these types. (Also see the **Catalog** and **Prototype** interfaces.)

ViewPoint clients must manage all file types that they use. Ranges of file types may be obtained from the Filing group.

14.2 Interface Items

root: NSFile.Type = ...;

The root file of the volume has this type. The root has children that are called (by convention) *catalogs*.

desktop, desktopCatalog: NSFile.Type = ...;

The desktop catalog contains all the desktops on a workstation. An individual desktop has the same type as the desktop catalog.

prototypeCatalog: NSFile.Type = ...;

The prototype catalog contains prototype files for each application. A prototype file is a blank application file that the user can make copies of, such as Blank Folder, or Blank Document. (See the **Prototype** interface.)

systemFileCatalog: NSFile.Type = ...;

The system file catalog contains system files, such as the bcds for an application, message files, font files, TIP files,and so forth. (See the **Catalog** interface.)

14.3 Index of Interface Items

15

BWSZone

15.1 Overview

BWSZone defines several zones, each with different characteristics, that ViewPoint clients may use, as appropriate.

15.2 Interface Items

All zones are created at boot time and exist for the duration of the boot session.

permanent: UNCOUNTED ZONE;

Permanent: PROCEDURE RETURNS [UNCOUNTED ZONE];

permanent is intended for nodes that are never deallocated. It has infinite threshold. **Permanent** returns **permanent**.

logonSession: UNCOUNTED ZONE;

LogonSession: PROCEDURE RETURNS [UNCOUNTED ZONE];

logonSession is intended for nodes that last for a logon/logoff session. **logonSession** is emptied of all nodes at each logoff (that is, **Heap.Flush**). **LogonSession** returns **logonSession**. **logonSession** is created at boot time and is flushed at logoff.

shortLifetime: UNCOUNTED ZONE;

ShortLifetime: PROCEDURE RETURNS [UNCOUNTED ZONE];

shortLifetime is intended for nodes that are allocated for a very short time, such as during a notification. **ShortLifetime** returns **shortLifetime**.

semiPermanent: UNCOUNTED ZONE;

SemiPermanent: PROCEDURE RETURNS [UNCOUNTED ZONE];

semiPermanent is intended for nodes that are allocated for a very long time but that might occasionally have to be expanded. **SemiPermanent** returns **semiPermanent**.

15.3 Index of Interface Items

16

Catalog

16.1 Overview

Catalog manipulates files called *catalogs* that are direct descendants of the root file on a NSFiling volume. Each catalog is uniquely identified by its file type. Files can be opened and created within a catalog. Catalogs can be opened, created, and enumerated.

Viewpoint creates a system file catalog and a prototype catalog (see the **Prototype** interface) at boot time. The system file catalog typically holds font files, TIP files, icon picture files, message files,and so forth.

16.2 Interface Items

16.2.1 Finding and Creating Files in a Catalog

```
GetFile: PROCEDURE [
    catalogType: NSFile.Type ← BWSFileTypes.systemFileCatalog,
    name: XString.Reader,
    readonly: BOOLEAN ← FALSE,
    session: NSFile.Session ← NSFile.nullSession]
    RETURNS [file: NSFile.Handle];
```

GetFile finds a file with name **name** in the catalog with type **catalogType**. If the file cannot be found, **NSFile.nullHandle** is returned.

```
CreateFile: PROCEDURE [
    catalogType: NSFile.Type ← BWSFileTypes.systemFileCatalog,
    name: XString.Reader,
    type: NSFile.Type,
    isDirectory: BOOLEAN ← FALSE,
    size: LONG CARDINAL ← 0,
    session: NSFile.Session ← NSFile.nullSession]
    RETURNS [file: NSFile.Handle];
```

CreateFile creates a file with the specified attributes (**name**, **type**, **isDirectory**, **size** in bytes) in the catalog with type **catalogType**.

16.2.2 Operating on Catalogs

```
Open: PROCEDURE [
    catalogType: NSFile.Type,
    session: NSFile.Session ← NSFile.nullSession]
    RETURNS [catalog: NSFile.Handle];
```

Opens the catalog with type catalogType. If the catalog cannot be opened, it returns NSFile.nullHandle.

```
Create: PROCEDURE [
    name: XString.Reader,
    catalogType: NSFile.Type,
    session: NSFile.Session ← NSFile.nullSession]
    RETURNS [catalog: NSFile.Reference];
```

Creates a catalog with the specified name and type. If the catalog already exists or cannot be created, it returns NSFile.nullReference. Note: Even though the file can be identified by type only, the name should be logical (such as "System Files") so that any tools written to manipulate catalogs can display these names.

```
Enumerate: PROCEDURE [proc: CatalogProc];

CatalogProc: TYPE = PROCEDURE [catalogType: NSFile.Type]
    RETURNS [continue: BOOLEAN ← TRUE];
```

Enumerate calls the client-supplied proc for each existing catalog or until proc returns FALSE.

```
beforeLogonSession: NSFile.Session;
```

beforeLogonSession is a session that can be used when calling a Catalog procedure before any user has logged on, such as at boot time. It is set to be the default session until a user logs on.

16.3 Index of Interface Items

17 Containee

17.1 Overview

Containee is an application registration facility. An application is a software package that implements the manipulation of one type of file. **Containee** is a facility for associating an application with a file type.

17.1.1 Background

All **NSFiles** have:

- a name
- a file type (LONG CARDINAL)
- a set of attributes, such as create date
- either:
  - content, such as a document
  - children that are also **NSFiles**, such as a folder

An **NSFile** that has children is often called a directory. Fine Point: An NSFile can actually have both content and children, that is ignored for now to simplify this discussion. Since the children of an **NSFile** can themselves have children, **NSFile** supports a hierarchical file system.

A ViewPoint desktop is backed by an **NSFile** that has children. Each child file of the desktop's **NSFile** is represented on the screen by an iconic picture.

Each application operates on **NSFiles** of a particular file type. For example, ViewPoint documents operate on **NSFiles** with file type of 4353. Each document icon is actually an **NSFile** of type 4353. Each application needs a way to register its ability to operate on files of a particular type. **Containee** is such a facility.

17.1.2 Containee.Implementation

An application's ability to operate on files of a particular type includes such operations as:

- Display of the iconic picture (full size and tiny).
- Open, performed when the user selects an icon and presses OPEN.
- Properties, performed when the user selects an icon and presses PROPS.

- Take the current selection, performed when the user drops an object onto an icon by COPYing or MOVEing a selected object to an icon.

An application registers itself by calling Containee.SetImplementation, supplying a file type and a Containee.Implementation. A Containee.Implementation is a record that contains two important procedures:

- A procedure for displaying an icon picture (Containee.PictureProc).
- A procedure for performing various operations on an icon, such as open, create a property sheet, and take the current selection (Containee.GenericProc).

This application registration allows the ViewPoint desktop implementation to be open-ended. The desktop implementation itself does not know how any file behaves. Rather it depends on applications registering their ability to operate on particular file types. The desktop implementation, at logon, simply enumerates the child files of the desktop's NSFile (using NSFile.List), obtaining the file type for each child. For each child file, the desktop implementation gets an application's Containee.Implementation by using the child file's file type (and Containee.GetImplementation) and then calls that application's Containee.PictureProc to actually display an icon picture. Similarly, when the user selects an icon on the desktop and presses OPEN, the desktop implementation uses the file type of the file at that place on the desktop to get the application's Containee.Implementation and then calls the application's Containee.GenericProc to get a StarWindowShell created. The implementations of Folders and File Drawers are similar to the desktop implementation in this respect.

17.1.3 Containee.Data

An application needs to distinguish one file from another. Two different documents may be the same file type, but probably have different names and different contents. Whenever an application's Containee.DisplayProc or Containee.GenericProc is called, the particular file being operated on by the user is passed to the procedure through the Containee.DataHandle parameter. A Containee.DataHandle is a pointer to a Containee.Data that is simply a record with an NSFile.Reference in it. An NSFile.Reference uniquely identifies a particular file and allows the application to utilize various NSFile file-accessing procedures for manipulating the file.

17.2 Interface Items

17.2.1 Items for Application Implementors

```
SetImplementation: PROCEDURE [NSFile.Type, Implementation]
  RETURNS [ Implementation];
```

SetImplementation associates an Implementation record with a particular file type and returns the previous Implementation that was associated with that file type. An application calls SetImplementation to register its ability to operate on files of a particular type.

When an application calls SetImplementation, it is convention to save the old implementation to backstop operations that the new implementation does not support. For example, most GenericProcs have an endcase that calls the old application's GenericProc for atoms it does not understand.

```
Implementation: TYPE = RECORD [
   implementors: LONG POINTER ← NIL,
   name: XString.ReaderBody ← XString.nullReaderBody,
   smallPictureProc: SmallPictureProc ← NIL,
   pictureProc:PictureProc ← NIL,
   convertProc: Selection.ConvertProc ← NIL,
   genericProc:GenericProc ← NIL ];
```

When an application registers its ability to operate on files of a particular type (i.e, calls SetImplementation), it supplies an Implementation record. The Implementation record defines the behavior of all files of that type.

implementors is provided for the convenience of clients that may want to associate some application-specific data with the Implementation record. **Note:** This data is one per application, not one per file.

name is a user-sensible name for the objects that the Implementation manipulates, such as "Document" or "Spreadsheet." This string typically comes from XMessage. The bytes of name are not copied--the storage for name must be allocated forever (which is easy to do using XMessage).

smallPictureProc is a procedure of type SmallPictureProc that returns a character. This procedure is describe below.

pictureProc is called whenever the file's full-sized icon picture needs to be painted. (See PictureProc.)

convertProc is called to convert the file into another form, such as an Interpress master. This procedure is used when the owner of the current selection is a container, such as a folder, and the selection is actually a file (row) in the container. The owner of the selection (i.e., the container implementation) may be called to convert the selected file (row), but only the application that implements that file's type can do the conversion. The convertProc allows the owner of the selection to pass the conversion request along to the application. The data parameter to the convertProc is a Containee.DataHandle. This convertProc does not need to be able to convert to a target type of file or fileType, but rather should call Containee.DefaultFileConvertProc for these target types. If the application does not perform conversion to any target types, Containee.DefaultFileConvertProc should be provided as the convertProc.

genericProc is where most of the application's real implementation resides. genericProc is called, for example, to open an icon, to produce a property sheet for an icon, to drop something on an icon, etc. See GenericProc.

```
SmallPictureProc: TYPE = PROCEDURE [
   data: DataHandle ← NIL,
   type: NSFile.Type ← ignoreType,
   normalOrReference: PictureState]
   RETURNS [smallPicture: XString.Character];
```

```
PictureState: TYPE = { garbage, normal, highlighted, ghost,
   reference, referenceHighlighted };

ignoreType: NSFile.Type = LAST[LONG CARDINAL];
```

The **SmallPictureProc** should return a character for the application, which should be obtained by passing a 13x13-bit icon picture to **SimpleTextFont.AddClientDefinedCharacter**. This character is used when the file is inside a folder. **normalOrReference** will be either **normal** or **reference**, and the appropriate small picture should be returned. The **SmallPictureProc** should try to use the **type** parameter first if it is not **Containee.ignoreType**. If it is **ignoreType**, the **SmallPictureProc** should use the **data** parameter. This change is necessary for allowing the reference icon application to work properly. Fine Point: The picture for **normalOrReference = reference/referenceHighlighted** will not normally be used by the folder application directly, but rather would be used by a generic reference icon application.

```
Data: TYPE = RECORD [
   reference: NSFile.Reference ← NSFile.nullReference ];

DataHandle: TYPE = LONG POINTER TO Data;

nullData:Data;
```

Data uniquely identifies a file. An application needs to distinguish one file from another. Two documents may be the same file type, but probably have different names and different contents. Whenever an application's **PictureProc** or **GenericProc** or **Implementation.convertProc** is called, the particular file being operated on by the user is passed to the procedure through the **DataHandle** parameter. An **NSFile.Reference** uniquely identifies a particular file and allows the application to utilize various **NSFile** file-accessing procedures for manipulating the file. **nullData** is a constant that should be used to represent a null **Containee.Data**.

```
GenericProc: TYPE = PROCEDURE [
   atom: Atom.ATOM,
   data:DataHandle,
   changeProc:ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [LONG UNSPECIFIED];
```

A **GenericProc** is a procedure supplied by an application as part of an **Implementation**. The **GenericProc** will be called to perform one of several operations that a user can invoke. **atom** tells the **GenericProc** what operation to perform. For example, when the user selects an icon and presses the OPEN key, the application's **GenericProc** is called with an **atom** of Open.

data identifies the particular **NSFile** to be operated on. The **NSFile**'s file type will be the one for which this application has registered its **Implementation**.

A **GenericProc** must return a value. The type of the return value depends on the **atom** passed in. Some atoms, their meaning to the **GenericProc**, and the expected return values are as follows:

Atom	Return Value and Meaning

CanYouTakeSelection

LONG POINTER TO BOOLEAN
If the application is willing to have the current selection dropped onto it, the **GenericProc** should return TRUE. This occurs when the user has selected something, pressed COPY or MOVE, and then selected one of this application's files. While the user has the mouse button down, the cursor changes to a question mark if the **GenericProc** returns FALSE; otherwise, the cursor stays the same and the icon picture highlights. This operation should be efficient and usually involves calling **Selection.CanYouConvert** or **Selection.HowHard** or **Selection.Query** to determine what **Selection.Targets** the selected object can be converted to. For example, the printing application's **GenericProc** returns TRUE if the current selection can be converted to an Interpress Master.
The **changeProc** need not be called for this atom.

CanYouTakeSelectionBackground

LONG POINTER TO BOOLEAN
This is only called after the **GenericProc** returns TRUE to **CanYouTakeSelection**. This atom asks if the **GenericProc** can support a background take using an encapsulated selection. The **CanYouTakeSelection** call is always called first to find out if the selection type is one that the **GenericProc** can take. The **CanYouTakeSelectionBackground** atom need not query the selection again; all that's needed is to return TRUE if it supports background take operations. If this atom returns TRUE, the caller will call **SelectionX.Encapsulate** (see the **Selection** chapter), do a FORK, and call the **GenericProc** with **TakeSelectionBackground** or **TakeSelectionCopyBackground**. If this atom returns FALSE, the client will not FORK and will do a foreground **TakeSelection** or **TakeSelectionCopy**.
The **changeProc** need not be called for this atom.

FreeMenu

None
The application should free the menu that was created for the Menu atom, if any. The **MenuData.MenuHandle** that was returned for the Menu atom will be passed as the **changeProcData**. This atom will not be passed to the GenericProc if the Menu atom returned NIL. The **changeProc** need not be called for this atom.

Menu

MenuData.MenuHandle
The application may create a menu. The menu will displayed by the system as a popup menu. This atom is passed when the user requests a popup menu for an icon, e.g. by pressing both mouse buttons simultaneously while the mouse is over an icon on the desktop or in a container window. If the application returns a menu, then it should be prepared to free the menu when the FreeMenu atom is passed to the GenericProc. The **changeProc** need not be called for this atom.

Open

StarWindowShell.Handle
The application should create a **StarWindowShell**. Usually, the content displayed in the **StarWindowShell** will be derived from the contents of the file. For example, the ViewPoint document

editor application displays the text and graphics contained in the file, thus making the file ready for viewing and/or editing.

Props	StarWindowShell.Handle The application should create a PropertySheet. Usually, the properties shown reflect some attributes of the file. For example, the Folder property sheet shows the name of the folder, how it is sorted, and how many objects it contains. These properties are all **NSFile** attributes of the file.
TakeSelection	LONG POINTER TO BOOLEAN The action performed for this atom is highly dependent on the particular application. This atom is passed when the user has selected something, pressed MOVE, then selected one of this application's files. For some applications, this means the selected object should be moved into this application; for example, the Folder application converts the selected object to a file and adds the file to the folder. For other applications, this means the selected object should be operated on in some application-specific fashion; for example, the printing application converts the selected object to an Interpress Master (file or stream) and then sends the master to a printer. The **GenericProc** should return TRUE if the operation was successful, FALSE otherwise.
TakeSelectionBackground	LONG POINTER TO BOOLEAN The same as **TakeSelection** except that the **GenericProc** is called from a background process so the **GenericProc** must use an encapsulated selection rather than the user's selection. To get the encapsulated selection, the **GenericProc** should raise the signal **GetContaineeDataContext** which will will be caught by the caller and will RESUME with a LONG POINTER TO SelectionX.**Saved** (see the **Selection** chapter). The **GenericProc** can then call SelectionX.**ConvertX** to get the value of the selection followed by Selection.**CopyOrMove**. The client should always call the **GenericProc** with **CanYouTakeSelection-Background** before calling with this atom.
TakeSelectionCopy	LONG POINTER TO BOOLEAN This atom has the same meaning as **TakeSelection**, except it corresponds to the COPY key being pressed rather than MOVE. Again, the meaning of this is highly application dependent.
TakeSelectionCopyBackground	LONG POINTER TO BOOLEAN This atom has the same meaning as **TakeSelectionCopy**, except it corresponds to the COPY key being pressed rather than MOVE. See **TakeSelectionBackground** for more details.

The **changeProc** must always be called, passing in **changeProcData** and an indication of which **NSFile** attributes have changed, if any. If the execution of the **GenericProc** causes any change to the **NSFile**'s attributes, calling the **changeProc** allows containers (such as Desktop, Folders) to update the display to reflect the changes. For example, when the **atom** is Props, the **GenericProc** must save the **changeProc** and return the StarWindowShell.**Handle** for the property sheet. Then later, if the user changes the file's name, for example, the application's **PropertySheet.MenuItemProc** gets control when the user is done and must then retrieve the **changeProc** and call it. (See the section on Usage/Examples for more detail.)

If the client's **GenericProc** is called with an **atom** that it does not recognize, it should call the previous **GenericProc** (using the old **Implementation** that was returned when it called **Containee.SetImplementation**). The original system-supplied **GenericProc** acts to backstop all possible atoms.

```
ChangeProc: TYPE = PROCEDURE [
   changeProcData: LONG POINTER ← NIL,
   data:DataHandle,
   changedAttributes: NSFile.Selections ← []
   noChanges: BOOLEAN ← FALSE];
```

A **ChangeProc** is a callback procedure that is passed to a **GenericProc**. It must always be called by the client regardless of whether an attribute of the file being operated has changed. The reason for always calling the **changeProc** is to allow deallocation of the **changeProcData**. The **noChanges** boolean indicates the effect on the relevant file's attributes. The **changeProcData** parameter must be correctly supplied even for the **noChanges** = TRUE case. This is used, for example, when the user changes the name of a file by using a property sheet. When the property sheet is taken down, the application changes the file's name and the **ChangeProc** that was passed to the **GenericProc** must then be called by the application. (See more detail in the section on Usage/Examples).

```
PictureProc: TYPE = PROCEDURE [
   data:DataHandle,
   window: Window.Handle,
   box: Window.Box,
   old, new: PictureState ];
```

```
PictureState: TYPE = {garbage, normal, highlighted, ghost, reference, referenceHighlighted};
```

A **PictureProc** is a procedure supplied by an application as part of an **Implementation**. The **PictureProc** is called whenever the desktop implementation needs to have the application's icon picture repainted or painted differently.

data identifies the particular **NSFile** whose picture should be painted. The **NSFile**'s file type will be the one for which this application has registered its **Implementation**. Even though all files of the same type will have the same **PictureProc** and therefore the same-shaped picture, each picture will differ because the name of the **NSFile** is often displayed on the picture. An application's **PictureProc** can obtain an **NSFile**'s name by using **NSFile** operations, but may more easily obtain it using **Containee.GetCachedName**. This is one of the primary intended uses for **GetCachedName**. (See the section on Attribute Cache).

window and **box** should be passed to any display procedures used to paint the icon picture, such as **Display.Bitmap** and **SimpleTextDisplay.StringIntoWindow**.

The **old** and **new** arguments describe the current and desired states of the icon picture. **garbage** is the unknown state. **PictureProc** will be called with **new = garbage** before moving or otherwise altering the icon; this lets an application remember an icon's placement. The application can thus continually update the icon (for example, to represent time-of-day) or can force a repaint by using **Window.Invalidate** (to change the shape of an

InBasket icon, for example), **normal** is the picture displayed when the icon is not selected. **highlighted** is the picture displayed when the icon is selected. **ghost** is the picture displayed when the icon is currently open. **reference** is the picture displayed to represent a remote file. **referenceHighlighted** is the highlighted version of **reference**. The desktop implementation will never use these last two states, but a generic reference icon application might.

DefaultFileConvertProc: Selection.ConvertProc;

DefaultFileConvertProc is a **Selection.ConvertProc** that knows how to convert to **Selection.Targets** of **file** and **fileType**. **DefaultFileConvertProc** should be called from an application's **Implementation.convertProc** for these targets, or should be provided as the application's **Implementation.convertProc** if the application has no **convertProc** of its own. No file-backed application's **convertProc** should need to worry about these target types.

GetContaineeDataContext: SIGNAL [dataHandle:DataHandle]
RETURNS [context: LONG POINTER];

This allows the client to pass some client data to the **GenericProc**. For certain atoms that the **GenericProc** and its clients agree upon, the **GenericProc** may raise this signal. The caller should catch the signal and resume with a long pointer to some mutually agreed upon data. One example of where this is used is in doing a background take. See the comments on the **TakeSelectionBackground** atom for how it uses **GetContaineeDataContext**. **Fine point:** GetContaineeDataContext is defined in ContaineeExtra.

17.2.2 Items for Application Consumers

These items would not ordinarily be used by an application implementation (provider), but rather by a consumer such as the Desktop or Folder implementation.

GetImplementation: PROCEDURE [NSFile.Type] RETURNS [Implementation];

GetImplementation returns the current **Implementation** for a particular file type.

17.2.3 DefaultImplementation

Containee supports a single global default **Implementation**. This default **Implementation** is used when the user operates on an **NSFile** for which no **Implementation** has yet been registered.

GetDefaultImplementation: PROCEDURE RETURNS [Implementation];

GetDefaultImplementation returns the current default **Implementation**.

SetDefaultImplementation: PROCEDURE [Implementation]
RETURNS [Implementation];

The default implementation provides a dummy display and appropriate "Sorry, Desktop is Unable to Open That Object" complaints in the absence of a particular implementation. Most clients will not call **SetDefaultImplementation**.

17.2.4 Attribute Cache

Clients often want to use several common **NSFile.Attributes**, but it is awkward to pass the attributes around in calls, because the attributes are long, of variable length, and frequently not needed by the called routine. Therefore, **Containee** provides a cache mechanism that can remember and supply popular attributes. Currently, the name and file type attributes are supported, as well as the run-time busy attribute. **Containee** decouples the management of in-memory copies of a file's name from parameter-passing arrangements.

```
GetCachedBusy: PROCEDURE [data: Containee.DataHandle] RETURNS [busy: BOOLEAN];
```

Returns TRUE if the file was made busy with **SetCachedBusy**, FALSE otherwise. 'Busyiness' is not actually an attribute stored with the file, but is actually a run-time bit maintained by Containee. See the **BusyIcon** chapter for a general interface to busy icons. **Fine point:** GetCachedBusy is defined in ContaineeExtra.

```
GetCachedName: PROCEDURE [data:DataHandle]
  RETURNS [name: XString.ReaderBody, ticket:Ticket];

GetCachedNameX: PROCEDURE [
  data:DataHandle,
  handle: NSFile.Handle ← NSFile.nullHandle,
  session: NSFile.Session ← NSFile.nullSession]
  RETURNS [name: XString.ReaderBody, ticket:Ticket];
```

GetCachedName returns the name attribute of the **NSFile** referred to by **data**. If the name is not in the cache, it is looked up and added to the cache. **ticket** must be returned (by using **ReturnTicket**) when the client is through with the name. The **ticket** is to prevent one client from changing the name while another is looking at it. **GetCachedNameX** is identical to **GetCachedName**, but takes a handle and a session. If **handle** is non-null, **Containee** will use the handle instead of opening its own handle if it needs to fetch attributes from the file. **session** is used for any filing operations including opening the file if necessary, which is done if **handle** is **nullHandle**. **handle** and **session** are needed if the client has **data** (or its parent) open in a session other than the default session. **Fine point:** GetCachedNameX is defined in ContaineeExtra.

```
GetCachedType: PROCEDURE [data:DataHandle]
  RETURNS [type:NSFile.Type];

GetCachedTypeX: PROCEDURE [
  data:DataHandle,
  handle: NSFile.Handle ← NSFile.nullHandle,
  session: NSFile.Session ← NSFile.nullSession]
  RETURNS [type:NSFile.Type];
```

GetCachedType returns the type attribute of the **NSFile** referred to by **data**. If the type is not in the cache, it is looked up and added to the cache. **GetCachedTypeX** is identical to **GetCachedName**, but takes a handle and a session. If **handle** is non-null, **Containee** will use the handle instead of opening its own handle if it needs to fetch attributes from the file. **session** is used for any filing operations including opening the file if necessary, which is done if **handle** is **nullHandle**. **handle** and **session** are needed if the client has **data** (or its parent) open in a session other than the default session **Fine point: GetCachedNameX** is defined in **ContaineeExtra. Fine point: GetCachedTypeX** is defined in **ContaineeExtra.**

InvalidateCache: PROCEDURE [data:DataHandle] ;

InvalidateCache clears any information about the **NSFile** from the cache. It is typically called when the attributes of an **NSFile** are changed by an application. An application rarely needs to call **InvalidateCache**, because calling the **ChangeProc** takes care of it.

InvalidateWholeCache: PROCEDURE ;

InvalidateWholeCache clears the entire cache. Information about all files is cleared.

ReturnTicket: PROCEDURE [ticket: Ticket];

ReturnTicket should be called after calling **GetCachedName**, when the client no longer needs the string.

```
SetCachedBusy: PROCEDURE [
   data: Containee.DataHandle,
   busy: BOOLEAN];
```

Mark the file "busy." The status of the file can later be queried with **GetCachedBusy**. 'Busyiness' is not an attribute stored with the file, but is a run-time status maintained by **Containee**. For a more general busy icon interface, see the **BusyIcon** chapter. **Fine point: SetCachedBusy** is defined in **ContaineeExtra.**

SetCachedName: PROCEDURE [data:DataHandle, newName: XString.Reader];

SetCachedName allows a client to change a cached name. Care should be taken to keep the filed name consistent with the cached name. An application rarely needs to call **InvalidateCache**, because calling the **ChangeProc** takes care of it.

SetCachedType: PROCEDURE [data:DataHandle, newType:NSFile.Type];

SetCachedType allows a client to change a cached type. Care should be taken to keep the filed type consistent with the cached type.

Ticket: TYPE[2];

A **Ticket** is returned when **GetCachedName** is called. When the client is done using the cached name, the ticket must be returned by calling **ReturnTicket**. This is to prevent one client from changing the name while another is looking at it.

17.3 Errors and Signals

```
Error: ERROR [msg: XString.Reader ← NIL, error: ERROR ← NIL,
    errorData: LONG POINTER TO UNSPECIFIED ← NIL];

Signal: SIGNAL [msg: XString.Reader ← NIL, error: ERROR ← NIL,
    errorData: LONG POINTER TO UNSPECIFIED ← NIL];
```

An application's **GenericProc** (and **PictureProc** and **ConvertProc**) should never assume that it has been called by a desktop, and therefore should never call such facilities as **Attention.Post** or **UserTerminal.BlinkDisplay**. (The application might be called by CUSP, for example.) Rather, the application should raise **Containee.Error** or **Signal** with an appropriate message. **Containee** will not catch these errors. The caller of the application's **GenericProc** should catch them and do the appropriate thing. In the typical case, the ViewPoint desktop calls the application's **GenericProc**; it catches the error and calls **Attention.Post** with the passed message. CUSP could catch the error and log the message in a log file.

msg is the message to display to the user. **error** is the actual lower-level error that occurred that caused **Error** or **Signal** to be raised. **errorData** points to any additional data that accompanied the lower-level error.

17.4 Usage/Examples

17.4.1 Sample Containee

The folder application is used as an example of a simple application that implements a particular file type.

```
-- Constants and global data

folderFileType: NSFile.Type = ...;
oldImpl: Containee.Implementation ← [];

-- Containee.Implementation procedures

FolderGenericProc: Containee.GenericProc =
<<[atom: Atom.ATOM,
   data: Containee.DataHandle,
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [LONG UNSPECIFIED]>>
   BEGIN
   SELECT atom FROM
      open => RETURN [MakeFolder[data, changeProc, changeProcData]];
      props => RETURN [MakePropertySheet[data, changeProc, changeProcData]];
      canYouTakeSelection => RETURN [IF CanITake[changeProc, changeProcData]
         THEN @true ELSE @false];
      canYouTakeSelectionBackground => RETURN [@TRUE]
      takeSelection, takeSelectionBackground =>
```

```
          RETURN [
            IF Take[data, move, changeProc, changeProcData,
                  atom = takeSelectionCopyBackground]
               THEN @true ELSE @false ];
      takeSelectionCopy, takeSelectionCopyBackground = >
         RETURN [
            IF Take[data, copy, changeProc, changeProcData,
               atom = takeSelectionCopyBackground ]
            THEN @true ELSE @false ];
      menu = >
         BEGIN
         run: XString.ReaderBody ← XString.FromSTRING ["AltOpen"L];
         name: XString.ReaderBody ← XString.FromSTRING ["Folder"L];
         title: MenuData.ItemHandle ← MenuData.CreateItem[
            zone: NIL, name: @name, proc: NIL];
         items: ARRAY[0..1) OF MenuData.ItemHandle ← [
            MenuData.CreateItem[zone: NIL, name: @run, proc: AltOpen]];
         menu: MenuData.MenuHandle ← MenuData.CreateMenu[
            zone: NIL, title: title, array: DESCRIPTOR[items]];
         RETURN [menu];
         END;
      freeMenu = >
         BEGIN
         menu: MenuData.MenuHandle ← changeProcData;
         MenuData.DestroyMenu [NIL, menu];
         RETURN[menu];
         END;
      ENDCASE = > RETURN [
         oldImpl.genericProc [atom, data, changeProc, changeProcData] ];
   END;

AltOpen: MenuData.MenuProc = {...};

CanITake: PROCEDURE [
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [yes: BOOLEAN] = {
   << Use Selection.CanYouConvert to see if the current selection can convert to a
file. If so, then return TRUE, else FALSE. >>
   };

MakeFolder: PROCEDURE [
   data: Containee.DataHandle,
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [shell: StarWindowShell.Handle] = {
   << Create and return a StarWindowShell containing a list of the files in this folder.
Use FileContainerShell.Create. >>
   };

MakePropertySheet: PROCEDURE [
   data: Containee.DataHandle,
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [psheet: StarWindowShell.Handle] = {
   << Create and return a property sheet, using PropertySheet.Create. >>
   };

Take: PROCEDURE [
   data: Containee.DataHandle,
   copyOrMove: Selection.CopyOrMove,
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [ok: BOOLEAN] = {
v: Selection.Value;
manager: LONG POINTER TO Selection.Saved ← @mgr;
mgr: Selection.Saved ← SelectionX.nullManager;>>
IF background THEN
   manager ← ContaineeExtra.GetContaineeDataContext[data];
<< If this is a background take, get the encasulated selection
    Convert the current selection to a file using SelectionX.Convert or
SelectionX.Enumerate with mgr, and copy or move that file into this folder. >>
   };

-- Initialization procedures

InitAtoms: PROCEDURE = {
   open ← Atom.MakeAtom["Open"L];
   props ← Atom.MakeAtom["Props"L];
   canYouTakeSelection ← Atom.MakeAtom["CanYouTakeSelection"L];
   canYouTakeSelectionBackground ←
   Atom.MakeAtom["CanYouTakeSelectionBackground"L];
   takeSelection ← Atom.MakeAtom["TakeSelection"L];
   takeSelectionCopy ← Atom.MakeAtom["TakeSelectionCopy"L];
   takeSelectionCopyBackground ←
   Atom.MakeAtom["TakeSelectionCopyBackground"L];
   menu ← Atom.MakeAtom["Menu"L];
   freeMenu ← Atom.MakeAtom["FreeMenu"L];
   };

SetImplementation: PROCEDURE = {
   newImpl: Containee.Implementation ← Containee.GetImplementation [
      folderFileType];
   newImpl.genericProc ← FolderGenericProc;
   oldImpl ← Containee.SetImplementation [ folderFileType, newImpl ];
   };

-- Mainline code
InitAtoms[];
SetImplementation[];
```

17.4.2 ChangeProc example

The folder property sheet is used to demonstrate a callback to a ChangeProc.

```
DataObject: TYPE = RECORD [
   fh: NSFile.Handle,
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL];

Data: TYPE = LONG POINTER TO DataObject;

MakePropertySheet: PROCEDURE [
   data: Containee.DataHandle,
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [pSheetShell: StarWindowShell.Handle] = {

   -- Pass changeProc to MakeItems through clientData.

   mydata: Data ← zone.NEW[DataObject ← [
      fh: NSFile.OpenByReference[@data.reference],
      changeProc: changeProc,
      changeProcData: changeProcData]];

   pSheetShell ← PropertySheet.Create [
      formWindowItems: MakeItems,
      menuItemProc: MenuItemProc,
      menuItems: [done: TRUE, cancel: TRUE, defaults: TRUE],
      title: XMessage.Get [...],
      formWindowItemsLayout: DoLayout,
      display: FALSE,
      clientData: mydata];
   };

MakeItems: FormWindow.MakeItemsProc = {
   -- Make property sheet items with calls to FormWindow.MakeXXXItem.
   };

MenuItemProc: PropertySheet.MenuItemProc = {
   <<[shell: StarWindowShell.Handle, formWindow: Window.Handle,
      menuItem: PropertySheet.MenuItemType, clientData: LONG POINTER]
      RETURNS [destroy: BOOLEAN ← FALSE]>>
   mydata: Data = clientData;
   SELECT menuItem FROM
      done => RETURN[destroy: ApplyAnyChanges[formWindow, mydata].ok];
      cancel => RETURN[destroy: TRUE];
      defaults => ...
      ENDCASE;
   RETURN[destroy: FALSE];
   };
```

```
ApplyAnyChanges: PROC [fw: Window.Handle, mydata: Data] RETURNS [ok: BOOLEAN] = {
   -- Collect any changes in the property sheet items.
   NSFile.ChangeAttributes [mydata.fh, ...];

   BEGIN -- Call the changeProc.
   data: Containee.Data ← [ NSFile.GetReference [mydata.fh] ];
   IF mydata.changeProc # NIL THEN
      mydata.changeProc[mydata.changeProcData, @data, changedAttributes];
   END;

   RETURN [ok: TRUE];
   };
```

17.4.3 Error and Signal Usage

This client catches an NSFile.Error and raises Containee.Error, passing along the ERROR and the NSFile.ErrorRecord:

```
message: XString.ReaderBody;
   errorRecord: NSFile.ErrorRecord;
   signal: --GENERIC-- SIGNAL ← NIL;
   file ← NSFile.OpenByReference [reference: ... !
      NSFile.Error = > {
   errorRecord ← error;
   signal ← LOOPHOLE[NSFile.Error, SIGNAL];
   GOTO ErrorExit}];
   < < Operate on the file.> >
   NSFile.Close[file];
   EXITS
      ErrorExit = > {
   message ← XString.FromSTRING["NSFile.Error"L];
   Containee.Error [msg: @message, error: signal, errorData: @errorRecord];
```

17.5 Index of Interface Items

18 ContainerCache

18.1 Overview

The **ContainerCache** interface provides the writer of a **ContainerSource** with a cache for the container's items. **ContainerCache** supports storing strings and client data with each item.

18.2 Interface Items

18.2.1 Cache Allocation and Management

Handle: TYPE = LONG POINTER TO **Object;**

Object: TYPE;

AllocateCache2: PROCEDURE [**useProcessAbort:** BOOLEAN ← TRUE] RETURNS [**Handle**];

AllocateCache2 returns a handle on a cache that can be filled with **BeginFill**. The client should call **ResetCache** before calling **BeginFill**. **useProcessAbort** indicates whether **Process.Abort** should be raised by **ContainerCache** when the fill process is aborted--for example, when the cache is destroyed while still filling. It is intended to accommodate clients that cannot properly handle **ABORTED**. **AllocateCache2** is actually in ContainerCacheExtra2.mesa.

AllocateCache: PROCEDURE RETURNS [**Handle**];

AllocateCache returns a handle on a cache that can be filled with **BeginFill**. The client should call **ResetCache** before calling **BeginFill**.

GetLength: PROCEDURE [**cache: Handle**] RETURNS [**cacheLength:** CARDINAL];

GetLength returns the number of items in the **cache**. **GetLength** is actually in ContainerCacheExtra.mesa.

ResetCache: PROCEDURE [**Handle**];

ResetCache clears the cache so that, for example, the cache can be refilled by calling **BeginFill**.

```
FreeCache: PROCEDURE [Handle];
```

Frees the resources used by a cache.

18.2.2 Filling the Cache

The client initially fills a cache with items by calling **BeginFill** with a **FillProc**. The **FillProc** adds items to the cache by repeatedly calling **AppendItem**.

```
FillProc: TYPE = PROCEDURE [cache: Handle]
  RETURNS [errored: BOOLEAN ← FALSE];
```

The client provides a **FillProc** to the **BeginFill** procedure. The **FillProc** should fill the cache by using **AppendItem**. **errored** is an indication of whether an error occurred during the filling of the cache (**errored** = TRUE).

```
BeginFill: PROCEDURE [
  cache: Handle,
  fillProc: FillProc,
  clients: LONG POINTER,
  fork: BOOLEAN ← TRUE ];
```

```
Clients: PROCEDURE [cache: Handle]
  RETURNS [clients: LONG POINTER];
```

BeginFill begins filling the cache. **fillProc** is called to add items to the cache. If **fork** is TRUE, then **fillProc** is forked as a separate process. **clients** is stored with the cache and may be retrieved by calling **Clients**.

```
CacheFillStatus: TYPE = {no, inProgress, inProgressPendingAbort,
  inProgressPendingJoin, yes, yesWithError, spare };
```

```
StatusOfFill: PROCEDURE [cache: Handle]
  RETURNS [CacheFillStatus];
```

StatusOfFill returns the current status of the cache fill. **yes** indicates that the fill has successfully completed; **no** means the cache has not been filled yet; **inProgress** indicates that the fill is running right now. **inProgressPendingAbort** indicates that an abort has been received but the **fillProc** has not yet returned. **inProgressPendingJoin**, **yesWithError**, and **spare** are not currently used.

18.2.3 Item Operations

```
ItemHandle: TYPE = LONG POINTER TO ItemObject;
```

```
ItemObject: TYPE;
```

```
AddData: TYPE = RECORD[
   clientData: LONG POINTER, -- TO ARRAY [0..0) OF WORD
   clientDataCount: CARDINAL,
   clientStrings: LONG DESCRIPTOR FOR ARRAY OF XString.ReaderBody];
```

An **AddData** record is passed to the **AppendItem**, **InsertItem**, and **ReplaceItem** procedures. **clientData** should contain any data that the client wants to cache with the item, usually some type of reference to the actual item. **clientDataCount** is the size (in words) of the **clientData**. **clientData** is copied into the cache; therefore the **clientData** should contain no pointers to other data. **clientStrings** should contain the strings to be displayed for the item. **clientStrings** are also copied into the cache, allowing the client to free them.

The standard use of **clientStrings** is to implement the **ContainerSource.StringOfItemProc**, which can be accessed efficiently by using **ItemNthString**. (See the section on item content operations for more details on accessing the contents of items.) **Caution:** There are restrictions on the total length of an item (strings plus client data) that may be added to a cache. Currently, no item should be longer than 512 bytes.

```
AppendItem: PROCEDURE [
   cache: Handle,
   addData: AddData]
   RETURNS [handle:ItemHandle];
```

AppendItem appends an item to the end of **cache**. It is usually called repeatedly from within a **FillProc**. **handle** is a pointer that can be used to access the new item.

```
DeleteNItems: PROCEDURE [
   cache: Handle,
   item: CARDINAL,
   nitems: CARDINAL ← 1];
```

DeleteNItems deletes one or more consecutive items from **cache**, starting at **item**. Fine point: Because the cache is maintained as a contiguous string of bits, this operation is likely to be slow compared to **AppendItem** and **GetNthItem**.

```
GetNthItem: PROCEDURE [cache: Handle, n: CARDINAL]
   RETURNS [ItemHandle];
```

GetNthItem returns the nth item in **cache**. The items are numbered from zero. It returns NIL if no such item exists. The **ItemHandle** returned is not guaranteed to be valid after any operation that modifies the cache (**DeleteNItems**, **InsertItem**, **ReplaceItem**). If the cache status is **inProgress** (if someone is in the process of filling the cache), **GetNthItem** does not return until the nth item has been appended to the cache or until the fill is complete.

```
InsertItem: PROCEDURE [
   cache: Handle,
   before: CARDINAL,
   addData: AddData]
   RETURNS [handle: ItemHandle];
```

InsertItem inserts an item in **cache**. The new item is inserted before the item **before**. Note that all the items after this item will be renumbered. Fine point: Because the cache is maintained as a contiguous string of bits, this operation is likely to be slow compared to **AppendItem** and **GetNthItem**.

```
ReplaceItem: PROCEDURE [
   cache: Handle,
   item: CARDINAL,
   addData: AddData]
   RETURNS [handle: ItemHandle];
```

ReplaceItem replaces the contents of **item** in **cache** with the information in **addData**. Fine point: This operation is implemented as **DeleteNItems** followed by **InsertItem**, and so is likely to be slow compared to **AppendItem** and **GetNthItem**.

18.2.4 Item Content Operations

```
ItemIndex: PROCEDURE [item: ItemHandle] RETURNS [index: CARDINAL];
```

Given the handle **item**, **ItemIndex** returns its **index** in the cache.

```
ItemClients: PROCEDURE [item: ItemHandle] RETURNS [clientData: LONG POINTER];
```

Returns the client data associated with **item**. If the client data passed in was **NIL**, **clientData** is **NIL**.

```
ItemClientsLength: PROCEDURE [item: ItemHandle] RETURNS [dataLength: CARDINAL];
```

Returns the length of the client data passed in with **item**.

```
ItemStringCount: PROCEDURE [item: ItemHandle] RETURNS [strings: CARDINAL];
```

Returns the number of client strings associated with **item**.

```
ItemNthString: PROCEDURE [item: ItemHandle, n: CARDINAL] RETURNS [XString.ReaderBody];
```

Returns the **n**th client string associated with **item**. This operation can be used to implement a **ContainerSource.StringOfItemProc**.

18.2.5 Marking Items in the Cache

Whenever items are deleted or inserted in a **ContainerCache**, all the items are renumbered. This allows a client to keep track of items by marking them. **ContainerCache** keeps track of the marked items across any changes to the cache. A **mark** is a handle on a cache item that tracks the item when the item number changes. This facility is handy for container source implementations that use **ContainerCache** and want to perform all the various combinations of moving and copying items within the source.

```
Mark: TYPE = LONG POINTER TO MarkObject;
MarkObject: TYPE;

SetMark: PROCEDURE [
    cache: ContainerCache.Handle, index: CARDINAL]
    RETURNS [mark: Mark];
    -- set a mark at index

IndexFromMark: PROCEDURE [mark: Mark]
    RETURNS [index: CARDINAL];
    -- get the current value of this mark

MoveMark: PROCEDURE [mark: Mark, newIndex: CARDINAL];
    -- allows the resetting of a mark without using a new one

FreeMark: PROCEDURE [mark: Mark];
    -- mark no longer needed
```

18.3 Usage/Examples

After the client allocates a cache, the client starts filling the cache by calling **BeginFill** with a **FillProc**. **BeginFill** immediately calls the **FillProc**. Inside the **FillProc**, the client usually does some kind of enumeration on the source backing (for example, if the source is backed by files, the client does an **NSFile.List**). For each item enumerated by the **FillProc**, the client builds the required strings for that item and then passes the strings along with any item data to **AppendItem**. The item data is usually some information that is needed to identify the item uniquely (for the file example, this might be a file ID). This process continues until all the items in the source have been enumerated, at which time the **FillProc** returns.

The call to **BeginFill** may indicate that the **FillProc** should be forked into a separate process. This allows the enumeration of the source's items to go on in the background, which is an advantage if the source has a large number of items. If the source is being displayed in a **ContainerWindow** while this background fill is taking place, the window displays each new item as it is appended to the cache. Fine point: **ContainerWindow** can display the items as they are added because **GetNthItem** will wait during the filling of the cache until the requested item is in the cache instead of returning with an indication that the requested item is not available.

Once the cache has been created, operations on the container source that owns the cache may cause items in the cache to become invalid. One way to bring the cache back into synch is to invoke **BeginFill** and rebuild the cache. If reenumerating the items in the source is expensive, items in the cache can be updated with the operations **DeleteNItems**, **InsertItem**, and **ReplaceItem**. The disadvantage of these operations is that they may cause performance degradation. Fine Point: The current implementation tries to maintain the cache as a contiguous series of strings of bits to minimize swapping. Using these operations may move large amounts of data around or fragment the cache data. If a large number of changes are to be made, it may pay to rebuild the cache.

Use of **ContainerCache** may not always be appropriate. In some cases, the structure of items in a source may be simple enough that a simple data structure may suffice to hold all the information necessary to respond to source operations.

18.3.1 Example of ContainerCache Use

The following example is taken from the implementation of **FileContainerSource**. It gives an example **FillProc** that uses **AppendItem** to build the cache.

```
ReaderSeq: TYPE = RECORD [SEQUENCE length: CARDINAL OF XString.ReaderBody];
ReaderSeqPtr: TYPE = LONG POINTER TO ReaderSeq;

WriterSeq: TYPE = RECORD [SEQUENCE length: CARDINAL OF XString.WriterBody];
WriterSeqPtr: TYPE = LONG POINTER TO WriterSeq;

FillCacheInBackground: ContainerCache.FillProc =
  <<[cache: Handle] RETURNS [errored: BOOLEAN ← FALSE]>>
  BEGIN
  fs: FS ← ContainerCache.Clients[cache]; -- get container source context
  parentHandle: NSFile.Handle;
  writers: WriterSeqPtr ← AllocateWriters [fs.columns.length];
  readers: ReaderSeqPtr ← z.NEW [ReaderSeq[fs.columns.length]];

  Enumerator: NSFile.AttributesProc =
    BEGIN
    itemData: ItemFileData;
    addData: ContainerCache.AddData;

    addData ← BuildRow [fs, writers, readers, @itemData, attributes];
    [] ← ContainerCache.AppendItem [cache, addData];
    RETURN;
    END;

  BEGIN
    parentHandle ← NSFile.OpenByReference [fs.parentReference];
    Process.SetPriority [Process.priorityBackground];
    NSFile.List [ directory: parentHandle, proc: Enumerator,
      selections: fs.selections, scope: fs.scope ];
    NSFile.Close [parentHandle];
  END;
  z.FREE [@readers];
  FreeWriters [writers];

  RETURN;
  END;

BuildRow: PROCEDURE [
  fs: FS,
  writers: LONG POINTER TO WriterSeq,
  readers: LONG POINTER TO ReaderSeq,
  itemData: ItemFileDataHandle,
  attributes: NSFile.Attributes]
```

```
RETURNS [addData: ContainerCache.AddData] =
BEGIN
attr: NSFile.Attribute;
ci: Containee.Implementation;

ci ← Containee.GetImplementation [attributes.type];
FOR i: CARDINAL IN [0..fs.columns.length) DO
   XString.ClearWriter [@writers[i]];
   -- Decide the type of column we have (passed in as Column info to
      FileContainerSource.Create) and call proper format proc to format attribute(s)
      into a string --
   WITH column: fs.columns[i] SELECT FROM
      attribute => {
         attr ← AttributeFromAttributeRecord [
            attributes, column.attr];
         column.formatProc [ci, attr, @writers[i]];};
      extendedAttribute => {
         attr ← ExtendedAttributeFromAttributeRecord [
            attributes, column.extendedAttr];
         column.formatProc [ci, attr, @writers[i]];};
      multipleAttributes =>
         column.formatProc [ci, attributes, @writers[i]];
      ENDCASE;
   ENDLOOP;

itemData↑ ← [id: attributes.fileID, type: attributes.type];

FOR i: CARDINAL IN [0..writers.length) DO
   readers[i] ← (XString.ReaderFromWriter [@writers[i]])↑;
   ENDLOOP;

addData ← [
   clientData: itemData,
   clientDataCount: SIZE[ItemFileData],
   clientStrings: DESCRIPTOR[readers]];

RETURN[addData];
END;
```

18.4 Index of Interface Items

19

ContainerSource

19.1 Overview

The Container interfaces (**ContainerSource**, **ContainerWindow**, **FileContainerSource**, **FileContainerShell**, and **ContainerCache**) provide the services needed to implement an application that appears as an ordered list of items to be manipulated by the user. ViewPoint Folders are a typical example of such an application. **ContainerWindow** provides the user interface for containers. It displays each item as a list of strings and handles selection highlighting, scrolling, and so forth. When a **ContainerWindow** is created, a record of procedures is passed in. **ContainerWindow** obtains the strings of each item by calling one of these procedures. **ContainerWindow** also performs user operations on items, such as open, props, delete, insert, take the current selection, and selection conversion by calling other procedures in the record. This record of procedures and their implementation is called a container source. A container source can be thought of as a supply (source) of items for a **ContainerWindow**. A container source is responsible for implementing container source operations on its underlying representation of the items in the source.

The **ContainerSource** interface contains the procedure TYPEs that make up the record of procedures that a container source must implement. These procedure definitions encompass all the operations that a source of items must be able to perform. **ContainerSource** also provides a place to save data specific to a particular container source.

The procedure TYPEs defined by **ContainerSource** fall into three categories:

- **ActOnProc**, **CanYouTakeProc**, **GetLengthProc**, and **TakeProc** are operations on the source as a whole.
- **ConvertItemProc**, **DeleteItemsProc**, **ItemGenericProc**, and **StringOfItemProc** are operations on the individual items within the source.
- **SetGlobalChangeProcProc**, **GetGlobalChangeProcProc**, **IsBusyProc**, **SetBusyProc**, **SetMarkProc**, **FreeMarkProc**, **IndexFromMarkProc**, and **MoveOrCreateMarkProc** are housekeeping kinds of operations that support background (concurrent) move and copy operations within a source.

Note that the items in a container must exhibit behavior similar to the behavior defined by the **Containee** interface, such as open, props, take selection, convert. However, also note that the **Containee** interface defines the behavior of **NSFiles**, whereas **ContainerSource** is totally independent of **NSFile**. The items in a container may be backed by anything. The **FileContainerSource** interface is an example of a container source that is backed by **NSFiles**. The ViewPoint Directory application contains examples of container sources that are backed by Clearinghouse entries (such as the Filing and Printing dividers) and by simple strings in virtual memory (such as a domain divider).

The **ContainerCache** interface provides a mechanism for caching the strings and item-specific data for the items in a container source. The implementor of a container source might find **ContainerCache** to be handy.

19.2 Interface Items

19.2.1 Handle, Procedures, and ProceduresObject

```
Handle: TYPE = LONG POINTER TO Procedures;

Procedures: TYPE = LONG POINTER TO ProceduresObject;

ProceduresObject: TYPE = RECORD [
   actOn: ActOnProc,
   canYouTake: CanYouTakeProc,
   columnCount: ColumnCountProc,
   convertItem: ConvertItemProc,
   deleteItems: DeleteItemsProc,
   getLength: GetLengthProc,
   itemGeneric: ItemGenericProc,
   stringOfItem: StringOfItemProc,
   take: TakeProc];

ContainerSourceExtra.Procedures: TYPE =
   LONG POINTER TO ContainerSourceExtra.ProceduresObject;

ContainerSourceExtra.ProceduresObject: TYPE = RECORD [
  canYouTakeX: CanYouTakeProcX,
  takeX: TakeProcX,
  setGlobalChangeProc: SetGlobalChangeProcProc,
  getGlobalChangeProc: GetGlobalChangeProcProc,
  isBusy: IsBusyProc,
  setBusy: SetBusyProc,
  setMark: SetMarkProc ← NIL,
  freeMark: FreeMarkProc ← NIL,
  indexFromMark: IndexFromMarkProc ← NIL,
  moveOrCreateMark: MoveOrCreateMarkProc ← NIL];
```

Handle identifies a particular container source. **Handle** is a pointer to a pointer (**Procedures**) to a record of procedures (**ProceduresObject**) that are implemented by the container source. A container source typically EXPORTs a **Create** procedure that return a **Handle**. This **Handle** is then passed to **ContainerWindow.Create**. Whenever **ContainerWindow** needs the container source to do something, it calls the appropriate procedure in the **ProceduresObject** by using **Handle↑↑**, and passing in the **Handle**. Note: Every procedure in the **ProceduresObject** takes a **Handle** as its first parameter. Fine Point:

Actually, **ContainerWindow** will call the **INLINE** procedures described in the INLINE section, which in turn call the procedures in the **ProceduresObject**.

Handle is a pointer to a pointer (rather than just a pointer to the **ProceduresObject**) to allow a container source to save data specific to the source. For example, a file-backed source would need to keep a pointer to the file. See the section on Usage/Examples for an explanation of how this is done.

ContainerSourceExtra.ProceduresObject are extra procedures to support concurency in containers. These procedures logically belong in the **ContainerSource.ProceduresObject**. No ContainerSourceExtra.Handle is provided because these procedures are auxiliary to the main **ContainerSource.Procedures**. A source that supports these operations will export a **Create** operation that returns a **ContainerSource.Handle** and a **ContainerSourceExtra.Procedures**. **ContainerWindowExtra4.CreateXX** takes both a **ContainerSource.Handle** and a **ContainerSourceExtra.Procedures**. This procedure must be used when the client wants to create a container that uses **ContainerSourceExtra.Procedures**.

19.2.2 Procedures That Operate on Individual Items

```
ItemIndex: TYPE = CARDINAL;

nullItem: ItemIndex = ItemIndex.LAST;
```

All the procedures that operate on individual items take a **Handle** and an **ItemIndex**. An **ItemIndex** is simply a **CARDINAL** that uniquely identifies an item in the source. Note: A container source is an *ordered* list of items. An **ItemIndex** of "n" indicates the "nth" item in the source. An **ItemIndex** of zero corresponds to the first source item. An **ItemIndex** should be thought of as a loose binding: the index of a particular item may change as a result of changes to the source. For example, if an item is deleted, all the items below it will be renumbered. **nullItem** is a constant used to represent no item or unknown item.

If concurrency is supported within the source, each of the procedures that take an **ItemIndex** as a parameter (**StringOfItemProc**, **ItemGenericProc**, **ConvertItemProc**, **DeleteItemsProc**, and **TakeProc/TakeProcX**) must be able to support a "between calls lock" on the source that lasts from the time the client calls **IndexFromMarkProc** to the time one of the following procedures is entered. See §19.2.5 for a discussion of **IndexFromItem**, §19.2.6 for a discussion of locking, and the example in §19.4.3 for one way to implement this locking.

```
StringOfItemProc: TYPE = PROCEDURE [
  source: Handle,
  itemIndex: ItemIndex,
  stringIndex: CARDINAL]
  RETURNS [XString.ReaderBody];
```

The source's **StringOfItemProc** should return the string **stringIndex** of item **itemIndex** in **source**. Each item's display is composed of strings, one for each column of the container window. For example, an open Folder shows four columns: the icon picture, the name, the size, and the date. **stringIndex** will be **IN [0..source.columnCount[])** (see also **ColumnCountProc** in the next section). If there is no such item (if **itemIndex** is greater than the number of items in the source, for example), **StringOfItemProc** should raise **Error[noSuchItem]**. **StringOfItemProc** is used extensively and its implementation should be efficient. If the source supports concurrency, this procedure must support the "between calls lock" convention. See §19.2.5 for more details.

```
ItemGenericProc: TYPE = PROCEDURE [
   source: Handle,
   itemIndex: ItemIndex,
   atom: Atom.ATOM,
   changeProc: ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [LONG UNSPECIFIED];
```

The source's **ItemGenericProc** is invoked to perform an operation on one of the items in the container. **itemIndex** indicates which item to operate on. The operation is specified by **atom**. Some of the typical atoms are: Open, Props, CanYouTakeSelection, TakeSelection, TakeSelectionCopy. This procedure is just like the **genericProc** that a **Containee.Implementation** must provide, see the **Containee** interface for a complete description of the atoms and their return values. **changeProc** must be called if the **ItemGenericProc** causes the source to change. **changeProc** and **changeProcData** are described in more detail below in the section on **changeProc** types.

If the source supports concurrency, this procedure must support the "between calls lock" convention. See §19.2.5 for more details.

```
ConvertItemProc: TYPE = PROCEDURE [
   source: Handle,
   itemIndex: ItemIndex,
   n: CARDINAL ← 1,
   target: Selection.Target,
   zone: UNCOUNTED ZONE,
   info: Selection.ConversionInfo ← [convert[]],
   changeProc: ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [value: Selection.Value];
```

The source's **ConvertItemProc** is invoked to convert one or more of the items in **source**, just as if the item was the current selection and **Selection.Convert** had been called. **itemIndex** indicates the first item to convert. **n** indicates how many consecutive items to convert. **target, zone, info,** and **value** are all identical to the parameters for **Selection.ConvertProc** (see the **Selection** interface). If n > 1, then **info** is the **enumeration** variant; otherwise, it is the **convert** variant. **changeProc** must be called if the **ConvertItemProc** causes the source to change, for example, when an item is moved out of the source. **changeProc** and **changeProcData** are described in more detail in the section on **changeProc** types.

If the source supports concurrency, this procedure must support the "between calls lock" convention. See §19.2.5 for more details.

```
DeleteItemsProc: TYPE = PROCEDURE [
   source: Handle,
   itemIndex: ItemIndex,
   n: CARDINAL ← 1,
changeProc: ChangeProc ← NIL,
changeProcData: LONG POINTER ← NIL];
```

The source's **DeleteItemsProc** is invoked to delete consecutive items from **source**. **itemIndex** is the first item to delete. **n** is the number of items to delete. **changeProc** must be called if the **DeleteItemsProc** causes the source to change, that is, if the deletion is successful. **changeProc** and **changeProcData** are described in more detail in the section on **changeProc** types.

If the source supports concurrency, this procedure must support the "between calls lock" convention. See §19.2.5 for more details.

19.2.3 Procedures That Operate on the Entire Source

```
ColumnCountProc: TYPE = PROCEDURE [ source: Handle] RETURNS [columns: CARDINAL];
```

The source's **ColumnCountProc** should return the number of columns in **source**, that is, the number of strings in each item. Fine point: typically, the number of columns is the same as COUNT [ContainerWindow.ColumnHeaders].

```
GetLengthProc: TYPE = PROCEDURE [ source: Handle]
RETURNS [length: CARDINAL, totalOrPartial: TotalOrPartial ← total];

TotalOrPartial: TYPE = {total, partial};
```

The source's **GetLengthProc** should return the total number of items currently in the source. This operation is performed often and should be efficient. Some container sources have indeterminate length until after an initial enumeration has completed (for example, clearinghouse enumerations). These sources may return [**totalOrPartial: partial**] while the initial enumeration is in progress. This lets the **ContainerWindow** display mechanism know that there are more items coming, while giving it some information along the way. Once a source knows how many items are in the source, (or for those sources that know right from the start how many items are in the source, (such as **NSFile**-backed sources), the **GetLengthProc** should return [**totalOrPartial: total**].)

```
ActOnProc: TYPE = PROCEDURE [ source: Handle, action: Action];

Action: TYPE = {destroy, reList, sleep, wakeup};
```

The source's **ActOnProc** is invoked to request some action of the source. **Action** indicates what the source should or can do.

destroy	The term **destroy** means that the source should destroy itself, freeing all storage and releasing all resources associated with the container source instance.
sleep	The term **sleep** means that the source should release whatever resources it can without losing information; it is a hint that the container source will not be used for a while.

wakeup	The term **wakeup** means that the source is going to be used and should resume its normal state, undoing whatever was done for **sleep**.
reList	The term **reList** means that the source should re-enumerate itself because its backing store has been changed.

```
CanYouTakeProc: TYPE = PROCEDURE [
   source: Handle,
   selection: Selection.ConvertProc ← NIL]
   RETURNS [yes: BOOLEAN];

CanYouTakeProcX: TYPE = PROCEDURE [
   source: Handle,
   background: BOOLEAN ← FALSE]
   RETURNS [yes: BOOLEAN];
```

The source's **CanYouTakeProc** is invoked to determine if the container source can take the current selection. **selection** is an obsolete parameter that is not used. If the **CanYouTakeProc** returns **yes** = TRUE, then the source's **TakeProc** may be called. **CanYouTakeProcX** takes a parameter **background** that asks if the source can take the current selection in the background. **CanYouTakeProcX** is exported by **ContainerSourceExtra**. Fine point: Supplying a **CanYouTakeProcX** is optional. However, if the source supplys the **CanYouTakeProcX**, it must also still provide a a **CanYouTakeProc**, even though the implementation of one may call the other.

This routine is intended to provide an efficient check on the compatibility of the objects being copied or moved. The common use of this routine is to provide feedback to the user. If a **CanYouTakeProc** returns TRUE, the client may choose to highlight the target. This is normally at the level of a file-type check. More elaborate checking is not necessary; for example, a file-backed container source would not want to check the source for protection or uniqueness violations. These should be handled by the **TakeProc**.

```
TakeProc: TYPE = PROCEDURE [
   source: Handle,
   copyOrMove: Selection.CopyOrMove,
   afterHint: ItemIndex ← nullItem,
   withinSameSource: BOOLEAN ← FALSE,
   changeProc: ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL,
   selection: Selection.ConvertProc ← NIL]
   RETURNS [ok: BOOLEAN];

TakeProcX: TYPE = PROCEDURE [
  source:Handle,
  copyOrMove: Selection.CopyOrMove,
  afterHint:ItemIndex ←nullItem,
  withinSameSource: BOOLEAN ← FALSE,
  changeProc:ChangeProc ← NIL,
  changeProcData: LONG POINTER ← NIL,
    mgr: SelectionX.Saved ← SelectionX.nullManager]
    RETURNS [ok: BOOLEAN];
```

```
beforeItemZero: ItemIndex = ItemIndex.LAST - 1;
```

The source's **TakeProc** is invoked to add items to the container source. **copyOrMove** tells the source whether to do a move or a copy of the current selection (which can be obtained by **Selection.Convert**). **afterHint** indicates the item the new item should be inserted after. Fine point: This is only a hint to the container source, since the ultimate position of the new item may depend on a sort order built in to the source. **afterHint** defaults to **nullItem**, which indicates that the caller doesn't care where the new item goes. If **afterHint** = **beforeItemZero**, the source should insert the new item before the first item. **changeProc** must be called if the **TakeProc** causes the source to change. **withinSameSource** = TRUE indicates to the source that the item(s) being moved or copied into the source are also in that same source; such as when the user moves or copies something from one place in a container to another place in the same container. This case usually involves some special case processing by the source (especially for move). **changeProc** and **changeProcData** are described in more detail in the next section. **selection** is an obsolete parameter that is not used. **ok** indicates whether the **TakeProc** was successful or not. The use of this routine is usually be preceded by a call to the source's **CanYouTakeProc**.

TakeProcX is the same as **TakeProc** with the addition of the **mgr** parameter that indicates the source of the items to be copied or moved. If **mgr** = **SelectionX.nullMgr**, the source is the current selection. Otherwise, **mgr** is an encapsulated selection that can be converted with **SelectionX.ConvertX**. See the **Selection** chapter for more information on encapsulated selections. Fine point: Supplying a **TakeProcX** is optional. However, if the source supplies the **TakeProcX**, it must also still provide a a **TakeProc**, even though the implementation of one may call the other.

If the source supports concurrency, these procedures must support the "between calls lock" convention. See §19.2.5 for more details.

19.2.4 ChangeProc Types

A source's **ConvertProc**, **DeleteItemsProc**, **ItemGenericProc**, and **TakeProc** all take a **ChangeProc** as an input parameter. This **ChangeProc** must be called by the source whenever any item or items in the source changes. This allows the **ContainerWindow** display code to keep the display up to date with the source. For example, a call to the source's **ItemGenericProc** with an atom of Props will cause a property sheet to be displayed for an item. If the user then edits, for example, the name of the item, and then closes the property sheet, the source must detect this change, update its backing, and call the **ChangeProc** that was passed into the **ItemGenericProc**. This **ChangeProc** (supplied by **ContainerWindow**) then causes the changed item(s) to be redisplayed.

```
ChangeProc: TYPE = PROCEDURE [
    changeProcData: LONG POINTER,
    changeInfo: ChangeInfo ];
```

A **ChangeProc** and **changeProcData** are passed to a source's **ConvertProc**, **DeleteItemsProc**, **ItemGenericProc**, and **TakeProc** . Since the **changeProcData** had to be allocated from someplace the **changeProc** must always be called, even if there were no changes to the source. The source must call the **ChangeProc** with the **changeProcData** and any **changeInfo**.

```
ChangeInfo: TYPE = RECORD [
   var: SELECT changeType: ChangeType FROM
      replace = > [item: ItemIndex],
      insert = > [insertInfo: LONG DESCRIPTOR FOR ARRAY OF EditInfo],
      delete = > [deleteInfo: EditInfo],
      all, noChanges = > NULL,
      ENDCASE ];
```

```
ChangeType: TYPE = { replace, insert, delete, all, noChanges};
```

ChangeInfo is passed to the **ChangeProc** to tell the display code exactly what changed. A container source can be smart and pass specific **ChangeInfo** (for example, "3 items were inserted after item 4 and 2 items were inserted after item 6" may be constructed with the **insert** variant), or be dumb and simply pass the **all** variant, which causes a total repaint of the container display. **replace** indicates that a single item has changed. **insert** indicates that one or more items have been inserted. **delete** indicates that one or more items have been deleted. **all** indicates that the entire source has been changed.

```
EditInfo: TYPE = RECORD [
   afterItem: ItemIndex,
   nItems: CARDINAL ];
```

EditInfo is used with the **insert** and **delete** variants of **ChangeInfo** to indicate how many items have been inserted or deleted, and where they were inserted at or deleted from.

19.2.5 Marks

A container source is defined as a sequence of items from **[0..length)**. Every time a item is inserted or deleted in the source, the rest of the items in the source are effectively renumbered. This causes trouble for **ContainerWindow**. If the user selects the fifth item, the container window must be able to continue to tie the selection to the object the user pointed at, even if other items have been added or deleted by concurrent operations.

We define *marks* to get around this problem. A mark is a handle on a item in the source that tracks that item when its item number changes.

The **ContainerCache** interface supports marks which can be used to implement ContainerSource marks. See the **ContainerCache** chapter for more information.

```
Mark: TYPE = LONG POINTER;
```

```
SetMarkProc: TYPE = PROCEDURE [
   source:Handle,
   index:ItemIndex]
   RETURNS [mark: Mark];
```

To mark an item, the client calls the **SetMarkProc** for a **source** and supplies an **index** to indicate the item to be marked. This creates a new **mark**. **Mark** and **SetMarkProc** are defined in **ContainerSourceExtra**.

```
IndexFromMarkProc: TYPE = PROCEDURE [
   source:Handle,
   mark: Mark,
   lockSource: BOOLEAN ← FALSE]
   RETURNS [index:ItemIndex];
```

IndexFromMarkProc takes a **mark** created with **SetMarkProc** or **MoveOrCreateMarkProc** and returns the current item index for mark as **index** .

A typical use of marks is to get the value of a mark and then call one of the source procs that takes an item index (i.e. **StringOfItemProc**, **ItemGenericProc**, **ConvertItemProc**, **DeleteItemsProc**, **SetBusy**, and **TakeProc/TakeProcX**). But with concurrency, the item index for a particular mark could change inbetween the time we call **IndexFromMarkProc** and when we call the source procedure. We establish a "between-calls lock" convention to address this problem. Calling **IndexFromMarkProc** with **lockSource** ← TRUE tells the container source to lock itself until client calls back to a procedure that takes an itemIndex or until the client unlocks the source with **SetBusy** (see §19.2.6). Thus the name "between-calls lock": the container source is only locked inbetween the call to **IndexFromMarkProc** and the next proc that takes an **itemIndex**. The other approach to use would be to lock the source, call **IndexFromMarkProc**, call source proc with the index, and unlock the source. This would result in the source being locked for the call to the source proc. This may be a undesirable if source proc might take a long time. Using **lockSource** unlocks the source again as soon as the source proc is called. **Important point**: if IndexFromMarkProc is called with **locksource** = TRUE and a source proc is not called, the client must call **SetBusy** to unlock the source.

IndexFrommarkProc is defined in **ContainerSourceExtra**.

```
FreeMarkProc: TYPE = PROCEDURE [
  source:Handle,
  mark: Mark];
```

FreeMarkProc frees **mark** when the client is done with it. **FreeMarkProc** is defined in **ContainerSourceExtra**.

```
MoveOrCreateMarkProc: TYPE = PROCEDURE [
   source:Handle,
   mark: Mark,
   newIndex:ItemIndex]
   RETURNS [newMark: Mark];
```

MoveOrCreateMarkProc is useful for pointing an existing **mark** at another item **newIndex**. If **mark** is NIL, the effect is the same as calling **SetMark**: a new mark is created. **newMark** is either the old mark updated, or the newly created mark . Even if **mark** is non-nil, the client must reassign **mark** to **newMark** because the value of **mark** may have changed. **MoveOrCreateMarkProc** is defined in **ContainerSourceExtra**.

19.2.6 ContainerSource locking and Busy routines

If a container source supports concurrency, it must support locking individual items and locking the entire container source. ContainerWindow locks individual items (or makes them 'busy') in response to user operations such as background copy out or background drop-on. The container source is responsible for knowing that a particular item is busy and responding with a busy status if queried.

```
SetBusyProc: TYPE = PROCEDURE [
  source:Handle,
  item:ItemIndex, -- if ItemIndex is nullItem, refers to whole source,
  newBusyState: BOOLEAN]
  RETURNS [succeeded: BOOLEAN];

IsBusyProc: TYPE = PROCEDURE [
  source:Handle,
  item:ItemIndex  -- if ItemIndex is nullItem, refers to whole source
  ]
  RETURNS [busy: BOOLEAN];
```

SetBusyProc changes the state of **item**. If **newBusyState** is TRUE, **item** should be made busy, if FALSE, **item** should be made unbusy. **IsBusyProc** gets the state of **item**. If the **item** is busy, it returns TRUE.

If **itemIndex** is **nullItem**, **Set/IsBusyProc** refer to the entire source rather than an individual item. When the entire source is locked, no items should be added or deleted to the source until the source is unlocked again. If the source is already locked when **SetBusyProc** is called, the call should wait until the source is unlocked.

Important implementation point: Because of the callback design of containers, **SetBusyProc** may be called in a nested fashion to lock a source: i.e., **ContainerWindow** may call **SetBusyProc[newBusystate: TRUE]** and later make the same call again with call with **newBusyState: FALSE** in the mean time to unlock the source. The container source should implement source locking such that nested calls to **SetBusyProc** from the same process do not lock, but a call from a different process will lock. Thus the first call from a given process locks the source for that process. §19.4.3 gives one example of how to implement a source locking scheme that will support these conventions. **Fine point:** if possible, this procedure should be relatively cheap. **ContainerWindow** will call it frequently: it will be called while tracking the selection for the user, for example.

19.2.7 Errors

A container source may raise **Error** or **Signal** as appropriate.

```
Error: ERROR [code: ErrorCode, msg: XString.Reader ← NIL,
   error: ERROR ← NIL, errorData: LONG POINTER TO UNSPECIFIED ← NIL];

Signal: SIGNAL [code: ErrorCode, msg: XString.Reader ← NIL,
   error: ERROR ← NIL, errorData: LONG POINTER TO UNSPECIFIED ← NIL];
```

A source's **ItemGenericProc** (and **ConvertItemProc** and **DeleteItemsProc**) should never assume that it has been called by a **ContainerWindow**, and therefore should never call such facilities as **Attention.Post** or **UserTerminal.BlinkDisplay**. (The application might be called by CUSP, for example.) Rather, the source should raise **ContainerSource.Error** or **Signal** with an appropriate message. The caller of the source's **ItemGenericProc** should catch these errors and do the appropriate thing. In the typical case, the **ContainerWindow** will call the source's **ItemGenericProc** and catch the error and call **Attention.Post** with the passed message. CUSP could catch the error and log the message in a log file. **msg** is the message to display to the user. **error** is the actual lower-level error that ocurred that caused **Error** or **Signal** to be raised. **errorData** points to any additional data that accompanied the lower level error.

```
ErrorCode: TYPE = MACHINE DEPENDENT {invalidParameters(0), accessError, fileError,
noSuchItem, other, last(15)};
```

invalidParameters	indicates that some parameters were invalid; for example, the **source** was not the correct type (the Procedures did not match).
accessError	indicates an attempt to perform an operation that violates the created access option (for sources that implement access controls).
fileError	indicates a file system error (for sources that are backed by files).
noSuchItem	A container source implementation should raise **Error[noSuchItem]** if one of the container source's procedures is called with an **ItemIndex** for an item that is not in the source.
other	may be raised to indicate any other problem.

Fine point: **Error** and **Signal** are **EXPORTed** by the **FileContainerSource** implementation since **ContainerSource** has no implementation.

19.2.8 Global change proc

```
GetGlobalChangeProcProc: TYPE = PROCEDURE [
  source: ContainerSource.Handle]
  RETURNS [
    changeProc:ChangeProc,
    data: LONG POINTER,
    window: Window.Handle];

SetGlobalChangeProcProc: TYPE = PROCEDURE [
  source:Handle,
  changeProc:ChangeProc,
  data: LONG POINTER,
  window: Window.Handle ← NIL];
```

If **SetGlobalChangeProcProc** is supplied by the source, container window will call this procedure during the **ContainerWindow.Create** call to supply a global change proc and data that the source can call anytime to update the container window. The container window may also give the source the **Window.Handle** for the container window so that clients of the source can get at the window if necessary. **window** may be **NIL** if the client of the source is not ContainerWindow. GetGlobalChangeProcProc allows any client of the source to get this information.

One example of where this is used is in the BusyIcon implementation. FileContainerSource provides a way for BusyIcon to find all the file-backed sources, and once BusyIcon finds the source a particular file is in, it calls the SetBusyProc to make the file, the calls GetGlobalChangeProcProc to get the change proc to allow it to update the container window display.

GetGlobalChangeProcProc and SetGlobalChangeProcProc are defined in ContainerSourceExtra.

19.2.9 INLINES

The following INLINE procedures are provided as a convenience to clients who wish to use object notation when calling a container source. ContainerWindow is the main client of these procedures.

```
ActOn: ActOnProc = INLINE {...};
CanYouTake: CanYouTakeProc = INLINE {...};
ColumnCount: ColumnCountProc = INLINE {...};
ConvertItem: ConvertItemProc = INLINE {...};
DeleteItems: DeleteItemsProc = INLINE {...};
GetLength: GetLengthProc = INLINE {...};
ItemGeneric: ItemGenericProc = INLINE {...};
StringOfItem: StringOfItemProc = INLINE {...};
Take: TakeProc = INLINE {...};
```

19.3 ContainerSource and concurrency

The ContainerSourceExtra interface defines a number of new container source procedures necessary to make concurrent move and copy work in a container source. If a particular type of container source does not create a ContainerSourceExtra.Procedures, the container window will realize from the absence of these procedures that it cannot support concurrency and will not start background operations to or from the source.

19.4 Usage/Examples

The reason that Handle is a pointer to a pointer (rather than just a pointer to the ProceduresObject) is to allow a container source to save data specific to the source. For example, a file-backed source would need to keep a pointer to the file. This is done in the following example.

19.4.1 ContainerSource Example

1. Declare a ContainerSource.ProceduresObject in the global frame of the module and fill it with the appropriate procedures.

```
mySourceProcs: ContainerSource.ProceduresObject ← [
   actOn: MyActOn,
   canYouTake: CanITake,
  columnCount: MyColumnCount,
  convertItem: ConvertMyItem,
  deleteItems: DeleteMyItems,
  getLength: GetMyLength,
  itemGeneric: MyItemGeneric,
  stringOfItem: StringOfMyItem,
  take: MyTake];
```

2. Declare a record that has a ContainerSource.Procedures (Procedures, not ProceduresObject!) as its first field and initialize this field to point to the ProceduresObject declared in the global frame. The rest of the record should contain whatever data the source needs in order to perform all the operations it will be requested to perform. Also declare a pointer to this record.

```
MySource: TYPE = LONG POINTER TO MySourceObject;

MySourceObject: TYPE = RECORD [
  procs: ContainerSource.Procedures ← @mySourceProcs,
  otherStuff: . . . ];
```

3. When creating the source, allocate the MySourceObject record and fill it with any relevant data. Return a pointer to the Procedures field of the record (@ms.procs below). Note: This return value is a pointer to a ContainerSource.Procedures, which is a ContainerSource.Handle.

```
Create: PUBLIC PROCEDURE [otherStuff: . . . ] RETURNS [source: ContainerSource.Handle] = {
  ms: MySource ← z.NEW [MySourceObject [otherStuff: otherStuff]];
  RETURN[@ms.procs];
  };
```

4. The first thing that every procedure in the ProceduresObject should do is LOOPHOLE the ContainerSource.Handle that was passed in into a pointer (MySource) to the source's data record (MySourceObject). After the LOOPHOLE, the fields of the source's data record can be directly accessed, e.g., ms.otherStuff. This all works because the first field in the source's data record is a Procedures. Note that the LOOPHOLE is actually performed in a procedure that also checks to be sure that the Procedures field of the passed source actually points to this source's procedures (IF source ↑ # @mySourceProcs THEN).

```
ActOnFile: ContainerSource.ActOnProc = {
  ms: MySource = ValidMySource[source];
  .
  . . . ms.otherStuff. . .
  .
  };

ValidMySource: PROCEDURE [source: ContainerSource.Handle] RETURNS [ms: MySource] = {
  IF source = NIL THEN ContainerSource.Error [invalidParameters] ;
  IF source ↑ # @mySourceProcs THEN ContainerSource.Error[invalidParameters];
  };
```

19.4.2 Errors and Signals

For example, this client catches an NSFile.Error and raises Containee.Error, passing along the ERROR and the NSFile.ErrorRecord:

```
message: XString.ReaderBody;
errorRecord: NSFile.ErrorRecord;
signal: --GENERIC-- SIGNAL ← NIL;
file ← NSFile.OpenByReference [reference: ... !
   NSFile.Error = > {
      errorRecord ← error;
      signal ← LOOPHOLE[NSFile.Error, SIGNAL];
      GOTO ErrorExit}];
-- Operate on the file.--
NSFile.Close[file];
EXITS
   ErrorExit = > {
      message ← XString.FromSTRING["NSFile.Error"L];
      ContainerSource.Error [
   code: fileError, msg: @message, error: signal, errorData: @errorRecord];
```

19.4.3 Source locking for concurrency

If a source supports background move and copy via the procedures defined in ContainerSourceExtra, it must also provide a means of locking the container source. As described in §19.2.5 and §19.2.6, this locking must support a number of conventions:

1. ContainerWindow must be able to lock the entire source by calling SetItemBusyProc . This lock prevents any items from being added or deleted in the source except by the calling process.
2. This locking must be reentrant: one process must be able to call by into the same source without getting monitor locked. Other processes must be locked out.
3. We must be able to support the "between calls" locking convention described in §19.2.5 under the discussion of IndexFromItemProc.

This can be a tricky set of constaints to satisfy. Number 2 is particularly tricky because conventional Mesa object locking doesn't do what we want: if we call an ENTRY procedure from within another ENTRY procedure, we deadlock. Number 1 implies the container source must lock itself whenever it modifies the source so that another call to lock the source will block until the modification is complete.

The example we present is taken from the FileContainerSource implementation. It provides a locking scheme that has proved easy to use and satisifies the requirements given above. This example shows code from from the implementation that has been modified slightly for clarity.

```
FSOps.FileSourceObject: TYPE = MACHINE DEPENDENT RECORD [
   procs (0:0..31): ContainerSource.Procedures,
   monitorLock (2:0..15): MONITORLOCK,   -- object lock field for the source implementation

   -- other source specific fields ...
```

```
  lock (21:0..79): RECORD [
    process (0:0..15): PROCESS ← NIL,
    entryCount (1:0..15): CARDINAL ← 0,
    lockedBetweenCalls (2:0..15): BOOLEAN ← FALSE,
    exiting(3:0..31): CONDITION]
  ];

Enter: PUBLIC PROC [fs: FS, how: EnterType ← normal] ;
Exit: PUBLIC PROC [fs: FS];

EnterType: TYPE = {
  normal,                  -- just do normal enter
  getBetweenCallsLock,  -- set betweenCalls boolean
  lockIfNotBetweenCalls -- if betweenCalls boolean not set, lock; otherwise nothing
  };
```

To implement our locking, we define two procedures: **Enter** and **Exit**. We also define a **lock** record as part of the source specific data to support these two procedures. Every procedure we would normally make an ENTRY procedure now calls **Enter** and **Exit** around the critical code. For any particular source **fs**, **Enter** allows a process into the monitor if there are no other processes running in the monitor. Once a particular process has the monitor, it can call **Enter** as many times as it wants so long as all **Enters** and **Exits** are paired. Any other process trying to get into the monitor waits until the running process has done its last **Exit**.

To support the "between calls" convention, we add a parameter to **Enter** to give some infomation about what our situation is. Most ordinary clients call with **how = normal**, which says 'I want to lock the source.'

Clients that take an **itemIndex** and must support "between call" locks call **Enter** with **how = lockIfNotBetweenCalls**. This says "if **IndexFromItemProc** was just called and the client wanted a between calls lock, don't lock the monitor again; otherwise, acquire the monitor." The logic here is that at the end of the ItemIndex procedure we call **Exit**. If we were not in the between calls case, the **Exit** unlocks the **Enter** at the beginning of the procedure. If we were in the between calls case, the **Exit** unlocks the **Enter** done in the **IndexFromItemProc**.
Finally, to set up the between calls state, **IndexFromItemProc** calls **Enter** with **how = getBetweenCallsLock**.

The implementation below give the implementation for **Enter** and **Exit**. Much of the logic is dedicated to making sure the between calls logic works correctly. **Exit** includes some debugging code to raise a signal if the number of **Exits** is more than the number of **Enters**. The entire module is an object monitor on the **monitorLock** field of the source object.

```
FileContainerSourceImpl: MONITOR LOCKS fs.monitorLock USING fs: FSOps.FS = BEGIN

Enter: PUBLIC ENTRY PROC [fs: FS, how: EnterType ← normal] =
  BEGIN ENABLE UNWIND => {};
  me: PROCESS = Process.GetCurrent[];
  SELECT fs.lock.process FROM
    me =>
      {IF ~(how = lockIfNotBetweenCalls AND fs.lock.lockedBetweenCalls) THEN {
        fs.lock.entryCount ← fs.lock.entryCount + 1;
```

```
            IF ~fs.lock.lockedBetweenCalls AND how = getBetweenCallsLock THEN {
              fs.lock.lockedBetweenCalls ← TRUE;
              };
            }
            ELSE fs.lock.lockedBetweenCalls ← FALSE
            RETURN};
        NIL => NULL;
        ENDCASE => {
          waitCount ← waitCount + 1;
          WHILE fs.lock.process # NIL DO WAIT fs.lock.exiting; ENDLOOP;
          waitCount ← waitCount-1;
          };

      fs.lock.process ← me;
      fs.lock.entryCount ← 1;
      fs.lock.lockedBetweenCalls ← how = getBetweenCallsLock;
      END;

UnbalancedFileContainerSourceLocks: SIGNAL = CODE;

Exit: PUBLIC ENTRY PROC [fs: FS] =
    BEGIN ENABLE UNWIND => {};
    me: PROCESS = Process.GetCurrent[];
    IF fs.lock.entryCount = 0 THEN SIGNAL UnbalancedFileContainerSourceLocks[] ;
    fs.lock.entryCount ← fs.lock.entryCount - 1;
    IF fs.lock.entryCount = 0 THEN
      {fs.lock.process ← NIL;
      NOTIFY fs.lock.exiting};
      --must not be BROADCAST; only the next process on the queue should be allowed
to run
    END;
```

ConvertFileItem is an example of a typical procedure that takes an ItemIndex and supports the between calls convention.

```
ConvertFileItem: ContainerSource.ConvertItemProc =
    BEGIN
    fs: FS = ValidFileSource[source];

    Enter[fs, lockIfNotBetweenCalls];          -- get a lock if we don't have one
    BEGIN ENABLE UNWIND => Exit[fs];
      IF inLock THEN Exit[fs];
        -- implement ConvertItem
      Exit[fs];
      END; -- enable
    END; -- ConvertFileItem
```

IndexFromMark also supports the between calls convention.

```
IndexFromMark: ContainerSourceExtra.IndexFromMarkProc = {
    fs: FS = ValidFileSource[source];
    IF lockSource THEN Enter[fs, getBetweenCallsLock];
    index ← IF mark # NIL THEN ContainerCache.IndexFromMark[mark]
      ELSE ContainerSource.nullItem;
    -- don't unlock: that will be done in callbacks.
    };
```

SetBusy has the dual function of locking individual items (which uses the between calls logic) and locking the entire source.

```
SetBusy: ContainerSourceExtra.SetBusyProc =
    BEGIN
    IF item # ContainerSource.nullItem THEN {
      Enter[fs, lockIfNotBetweenCalls];
      BEGIN ENABLE UNWIND = > Exit[fs];
        -- set busy status for item
        Exit[fs, 2100];
        END;-- enable
      }
    ELSE -- lock source
      IF newBusyState THEN Enter[fs] ELSE Exit[fs]};
    END;
```

19.5 Index of Interface Items

20

ContainerWindow

20.1 Overview

The **ContainerWindow** interface supports the creation of ViewPoint-like container windows. A container window provides a user interface that operates on a list of objects. The objects are displayed in rows. Each container window has one or more columns, with all rows displaying the same number of columns.

The **ContainerWindow** implementation maintains the display and manages user-invoked actions such as scrolling, selection, notifications, open within, show next/previous, and so forth. **ContainerWindow** takes a body window, a **ContainerSource**, and a specification of the columns and makes the window behave like a container. **Note:** This interface does not depend on **NSFile**: the objects represented by rows in the container do not have to be backed by **NSFiles**.

20.2 Interface Items

20.2.1 Create and Destroy a ContainerWindow

```
Create: PROCEDURE [
   window: Window.Handle,
   source: ContainerSource.Handle,
   columnHeaders: ColumnHeaders,
   firstItem: ContainerSource.ItemIndex ← 0]
   RETURNS [ regularMenuItems, topPusheeMenuItems: MenuData.ArrayHandle];

CreateX: PROCEDURE [
   window: Window.Handle,
   source: ContainerSource.Handle,
   columnHeaders: ColumnHeaders,
   firstItem: ContainerSource.ItemIndex ← 0,
   access: Access ← fullAccess]
   RETURNS [ regularMenuItems, topPusheeMenuItems: MenuData.ArrayHandle];

CreateXX: PROCEDURE [
   window: Window.Handle,
   source: ContainerSource.Handle,
   sourceX: ContainerSourceExtra.Procedures,
   columnHeaders:ColumnHeaders,
   firstItem: ContainerSource.ItemIndex ←0,
   access:Access ←fullAccess]
   RETURNS [regularMenuItems, topPusheeMenuItems: MenuData.ArrayHandle];

ColumnHeaders: TYPE = LONG DESCRIPTOR FOR ARRAY OF ColumnHeaderInfo;

ColumnHeaderInfo: TYPE = RECORD [
   width: CARDINAL,
   wrap: BOOLEAN,
   heading: XString.ReaderBody];

Access: TYPE = PACKED ARRAY AccessType OF BooleanFalseDefault;

BooleanFalseDefault: TYPE = BOOLEAN ←FALSE;

AccessType: TYPE = {open, dropOn, convert, add, delete, props};

fullAccess: Access = ALL [TRUE];

readOnlyAccess: Access = [open: TRUE, convert: TRUE, props: TRUE];

dividerAccess: Access = [open: TRUE, dropOn: TRUE, convert: TRUE, props: TRUE];
```

Create turns an ordinary window into a container window. **window** must be a **StarWindowShell** body window. **source** supplies a source of items to be displayed and manipulated (see the **ContainerSource** and **FileContainerSource** interfaces).

CreateX is just like **Create**, but with the additional **access** parameter. **ContainerWindow** displays an appropriate message to the user if s/he tries to do something for which proper access is not provided. **open, delete,** and **props** access give the user the capability to open icons, delete them, or open a property sheet on them. **dropOn** allows the user to drop something on an item, **convert** controls whether the ContainerWindow supports **Selection.Convert** (and thus copy and move out). **add** controls whether the user is allowed to add anything to the container display itself.
CreateX, Access, AccessType, fullAccess, readOnlyAccess, and dividerAccess are defined in ContainerWindowExtra3.mesa.

CreateXX is just like **Create** and **CreateX**, but adds the **sourceX** parameter to provide extra container source procedures needed for concurrency. (See the **ContainerSource** chapter for more information on concurrency in containers.) **CreateXX** is defined in ContainerWindowExtra4.mesa.

columnHeaders describes the column widths and supplies column headings. The columns will be displayed in the order given by this array. For each column, **width** is the number of bits the column should take, and **heading** is a string that will be displayed at the top of the column. **wrap** indicates what to do when a string that the container window wants to display is wider than **width**. If **wrap** = TRUE , the string should be wrapped around, otherwise, it will be truncated. Fine Point: **columnHeaders** is copied by **Create**, so this structure may be in the client's local frame.

firstItem indicates the item that should be displayed first when the container window is initially displayed.

regularMenuItems and **topPusheeMenuItems** are the menu items that the container window needs to have in the **StarWindowShell**. They should be added (by the client) to the menu that is installed in the **StarWindowShell** which this container window is a part of (these contain menu items such as Show Next and Show Previous).

```
Destroy: PROCEDURE [window: Window.Handle];
```

Destroys the data associated with the container window. Does *not* destroy the window itself. May raise **Error [notAContainerWindow]**.

20.2.2 Item operations

The individual containees in a container window are referred to as *items* (from **ContainerSource.ItemIndex**) They are sequentially numbered starting with zero.

```
DeleteAndShowNextPrevious: PROCEDURE [
   window: Window.Handle,
   item: ContainerSource.ItemIndex,
   direction: Direction ← next];
```

```
DeleteAndShowNextPrevious: PROCEDURE [
   window: Window.Handle,
   item: ContainerSource.ItemIndex,
   direction: Direction ← next]
   RETURNS [newOpenShell: StarWindowShell.Handle];
```

```
Direction: TYPE = {next, previous};
```

DeleteAndShowNextPrevious deletes **item** from the container source and the display, then displays the next or previous item. When this proc is called, the container window shell is expected to be on top. In particular, the shell of the item named in the item parameter should have been destroyed. So to implement this, if this item is opened within the container window, the client should call **StarWindowShell.Pop** until the shell returned from that call is equal to the container window shell. The second **DeleteAndShowNextPrevious** is defined in ContainerWindowExtra2.mesa. It is identical to the first one, but additionally returns the shell just opened. May raise **Error[notAContainerWindow]** or **Error[noSuchItem]**.

```
GetOpenItem: PROCEDURE [window: Window.Handle]
   RETURNS [item: ContainerSource.ItemIndex ← ContainerSource.nullItem];
```

Returns the item that is currently open within the container. If no item is open, returns **ContainerSource.nullItem**. May raise **Error[notAContainerWindow]**.

GetSelection: PROCEDURE [window: Window.Handle]
  RETURNS [first, lastPlusOne: ContainerSource.ItemIndex];

Returns the items currently selected in the ContainerWindow. first = last = ContainerSource.nullItem means there is no selection.

SelectItem: PROCEDURE [window: Window.Handle,
  item: ContainerSource.ItemIndex];

Selects the specified item and implicitly calls MakeItemVisible. MakeItemVisible is in a friends-level interface. Note: MakeItemVisible Forces item to be visible in window. If there is more than a screenful of items left following item, it is put at the top of the window. If less than a screenful remains, item is put at the bottom of the window with as many items as will fit before it. May raise Error[notAContainerWindow] or Error[noSuchItem].

20.2.3 Operations on a ContainerWindow

IsIt: PROCEDURE [window: Window.Handle] RETURNS [yes: BOOLEAN];

Returns TRUE if the window passed in is a ContainerWindow.

GetSource: PROCEDURE [window: Window.Handle]
  RETURNS [source: ContainerSource.Handle];

Returns the ContainerSource associated with this window. May raise Error[notAContainerWindow]. SetSource allows the client to change the source and the SourceModifyProc allows the client to modify the source.

KeepWindowOpen: PROCEDURE [window: Window.Handle];
WindowCanClose: PROCEDURE [window: Window.Handle];

KeepWindowOpen prevents the use from closing the container window. If the user tries to close the window, the message "Can't close that container while background operations are going on inside it." is posted. WindowCanClose allows the user to close the window again. In BWS 4.3, these procedures in are ContainerWindowExtra6.

SetSource: PROCEDURE [
  window: Window.Handle, newSource: ContainerSource.Handle]
  RETURNS [oldSource: Handle];

SourceModifyProc: TYPE = PROCEDURE [
  window: Window.Handle, source: ContainerSource.Handle]
  RETURNS [changeInfo: ChangeInfo];

ModifySource: PROCEDURE [window: Window.Handle, proc: SourceModifyProc];

ModifySource calls the source modification proc from within its monitor.

Update: PROCEDURE [window: Window.Handle];

Called when the correspondence between the source and the display is invalid. Items in the display will be redisplayed to reflect any changes in the source. May raise

Error[notAContainerWindow]. Fine Point: Clients will not normally need to call this routine unless they manipulate the source directly. All user-initiated operations on a **ContainerWindow** cause the display to be updated automatically.

20.2.4 Errors

Error: ERROR [code: ErrorCode];

ErrorCode: TYPE = MACHINE DEPENDENT {notAContainerWindow(0), noSuchItem, last(7)};

Any operations that operate on a container window may raise this error. **notAContainerWindow** is raised if the window passed in is not a container window (i.e., was not passed to **Create**). **noSuchItem** may be raised if an operation specifies a non-existent item.

20.3 Usage/Examples

The following example is taken from the implementation of the FileContainerShell interface. It illustrates the steps involved in creating a container window: creating a container source, creating a **StarWindowShell**, creating a body window inside the shell, creating the container window, and finally merging the menu items returned by ContainerWindow.Create with its own menu commands and installing those commands in the shell. It also gives a sample **StarWindowShell** transition procedure that will destroy the container source and the container window.

```
-- From FileContainerShellImpl.mesa

MenuItemSeq: TYPE = RECORD [
  SEQUENCE length: CARDINAL OF MenuData.ItemHandle];

Create: PUBLIC PROCEDURE [
  file: NSFile.Reference,
  columnHeaders: ContainerWindow.ColumnHeaders,
  columnContents: FileContainerSource.ColumnContents,
  regularMenuItems, topPusheeMenuItems: MenuData.ArrayHandle ← NIL,
  scope: NSFile.Scope ← [],
  position: ContainerSource.ItemIndex ← 0,
  options: FileContainerSource.Options ← []]
  RETURNS [shell: StarWindowShell.Handle] =

  BEGIN
  body: Window.Handle ← NIL;
  source: ContainerSource.Handle ← NIL;
  cwRegularMenuItems, cwTopPusheeMenuItems: MenuData.ArrayHandle;
  mergedMenuItems: LONG POINTER TO MenuItemSeq ← NIL;
  menu: MenuData.MenuHandle;
  name: XString.ReaderBody;
  ticket: Containee.Ticket;
  data: Containee.Data ← [file];
  type: NSFile.Type;
  smallPicture: XString.Character;
```

```
IF file = NSFile.nullReference THEN RETURN [ [NIL] ];
source ← FileContainerSource.Create [
   file: file,
   columns: columnContents,
   scope: scope,
   options: options];

[name, ticket] ← Containee.GetCachedName [@data];
type ← Containee.GetCachedType[@data];
smallPicture ← Containee.GetImplementation[type].smallPicture;

shell ← StarWindowShell.Create [
   name: @name,
   namePicture: smallPicture,
   sleeps: FALSE,
   transitionProc: DestroyProc ];

Containee.ReturnTicket [ticket];

body ← StarWindowShell.CreateBody [sws: shell, box: [[0,0],[700, 29999]]];

[cwRegularMenuItems, cwTopPusheeMenuItems] ← ContainerWindow.Create [
   window: body,
   source: source,
   columnHeaders: columnHeaders,
   firstItem: position];

mergedMenuItems ← MergeMenuArrays [cwRegularMenuItems, regularMenuItems];
IF mergedMenuItems # NIL THEN
   BEGIN
   menu ← MenuData.CreateMenu [
      zone: StarWindowShell.GetZone[shell],
      title: NIL,
      array: DESCRIPTOR[mergedMenuItems],
      copyItemsIntoMenusZone: TRUE ];
   StarWindowShell.SetRegularCommands [shell, menu];
   z.FREE[@mergedMenuItems];
   END;

mergedMenuItems ← MergeMenuArrays [cwTopPusheeMenuItems,
topPusheeMenuItems];
menu ← MenuData.CreateMenu [
   zone: StarWindowShell.GetZone[shell],
   title: NIL,
   array: DESCRIPTOR[mergedMenuItems],
   copyItemsIntoMenusZone: FALSE ];
StarWindowShell.SetTopPusheeCommands [shell, menu];
RETURN [shell];
END;
```

```
DestroyProc: StarWindowShell.TransitionProc =
<<[sws: StarWindowShell.Handle, state: StarWindowShell.State]>>
  BEGIN
  IF state = dead THEN {
    cw: Window.Handle ← GetContainerWindow[sws];
    source: ContainerSource.Handle ← GetContainerSource[sws];
    ContainerSource.ActOn [source, destroy];
    ContainerWindow.Destroy[cw]; };
  RETURN;
  END;

MergeMenuArrays: PROC [itemArray1, itemArray2: MenuData.ArrayHandle]
RETURNS [mergedSeq: LONG POINTER TO MenuItemSeq] =
  BEGIN
  i: CARDINAL ← 0;
  IF itemArray1 = NIL AND itemArray2 = NIL THEN RETURN[NIL];
  mergedSeq ← z.NEW [MenuItemSeq[itemArray1.LENGTH + itemArray2.LENGTH]];
  FOR j: CARDINAL IN [0..itemArray1.LENGTH) DO
    mergedSeq[i] ← itemArray1[j];
    i ← i + 1;
    ENDLOOP;
  FOR j: CARDINAL IN [0..itemArray2.LENGTH) DO
    mergedSeq[i] ← itemArray2[j];
    i ← i + 1;
    ENDLOOP;
  RETURN[mergedSeq];
  END;
```

20.4 Index of Interface Items

21

Context

21.1 Overview

In performing various functions, an application may wish to save and retrieve state from one notification to the next. This is an immediate consequence of the notification scheme, for a tool cannot keep its state in the program counter without stealing the processor after responding to an event. Thus the application must explicitly store its state in data. Because most notification calls to the application provide a window handle, it is natural to associate these *contexts* with windows. The context mechanism provides an alternative to the application's having to build its own associative memory to retrieve its context, given a window handle.

Typically, an application obtains a unique **Type** for its context data by calling **UniqueType** in the startup code for the application. Whenever a window is created, the client allocates some context data and calls **Create** to associate that data with the window. Whenever the client is called to perform some operation on the window (for example, to display the contents of the window or to handle a notification), it calls **Find** to retrieve the data saved with the window. Finally, when the window is being destroyed, the client (orViewPoint) calls **Destroy**, which calls the client's **DestroyProcType** to give the client an opportunity to free the data.

21.2 Interface Items

21.2.1 Creating/Destroying a Context

```
UniqueType: PROCEDURE RETURNS [type: Type];
```

The procedure **UniqueType** is called if a client needs a unique **Type** not already in use either by ViewPoint or by another client. If no more unique **types** are available, the **ERROR** **Error[tooManyTypes]** is raised.

```
Create: PROCEDURE [
  type: Type, data: Data, proc: DestroyProcType, window: Window.Handle];
```

The procedure **Create** creates a new context of type **type** that contains **data**. The context is associated with **window**; it is said to "hang" on the window. If **window** already has a context of the specified type, it raises the ERROR **Error[duplicateType]**. If the **window** is NIL, it raises the ERROR **Error[windowIsNIL]**. The **proc** is supplied so that when the window is destroyed, all of the context can be destroyed (deallocated).

Type: TYPE = MACHINE DEPENDENT{
 all(0), first(1), lastAllocated(37737B), last(37777B)};

Type is unique for each client of the context mechanism. An argument of this type is passed to most of the procedures in this interface so that the correct client data can be identified.

Data: TYPE = LONG POINTER TO UNSPECIFIED;

Data is the value that a client may associate with each window. It is typically a pointer to a record containing the client's state for some window.

DestroyProcType: TYPE = PROCEDURE [Data, Window.Handle];

A **DestroyProcType** is passed to **Create** so that the client can be notified when the context should be destroyed. This may be the result of the window being destroyed.

Destroy: PROCEDURE [type: Type, window: Window.Handle];

The procedure **Destroy** destroys a context of a specific **type** on **window**. If the context exists on the window, it calls the **DestroyProcType** for the context being destroyed.

DestroyAll: PROCEDURE [window: Window.Handle];

The procedure **DestroyAll** destroys all the contexts on **window**. Fine point: **DestroyAll** can be very dangerous because ViewPoint keeps its window-specific data in contexts on the window. **DestroyAll** should not be used except in special circumstances. It is called by the routines that destroy windows.

NopDestroyProc: DestroyProcType;

The procedure **NopDestroyProc** does nothing. It is provided as a convenience to clients that do not want to create their own do-nothing **DestroyProcType** to pass to **Create**.

SimpleDestroyProc: DestroyProcType;

The procedure **SimpleDestroyProc** merely calls the system heap deallocator on the **data** field. It is provided for clients whose context data is a simple heap node in the system zone.

21.2.2 Finding a Context on a Window

Find: PROCEDURE [type: Type, window: Window.Handle] RETURNS [Data];

The procedure **Find** retrieves the **data** field from the specified context for **window**. NIL is returned if no such context exists on the window.

FindOrCreate: PROCEDURE [
type: Type, window: Window.Handle, createProc: CreateProcType] RETURNS **[Data];**

The procedure **FindOrCreate** resolves the race that exists when creating new contexts in a multi-process environment. If a context of type **type** exists on **window**, it returns the context's **data**; otherwise, it creates a context of **type** by calling **createProc** and then returns **data**. If the **window** is NIL, it raises the ERROR **Error[windowIsNIL]**.

CreateProcType: TYPE = PROCEDURE RETURNS **[Data, DestroyProcType];**

CreateProcType is used by **FindOrCreate**. The procedure passed in as an argument to **FindOrCreate** is called to create a context only if a context of the appropriate type cannot be found.

Set: PROCEDURE **[type: Type, data: Data, window: Window.Handle];**

The procedure **Set** changes the actual data pointer of a context. Subsequent **Finds** will return the new data. **Note:** The client can change the data that the data field of a context points to at any time. This could lead to race conditions if multiple processes are doing **Finds** for the same context and modifying the data. It is the client's responsibility to MONITOR the data in such cases. If the **window** is NIL, it raises the ERROR **Error[windowIsNIL]**.

21.2.3 Acquiring/Releasing the Context

Acquire: PROCEDURE **[type: Type, window: Window.Handle]** RETURNS **[Data];**

The procedure **Acquire** retrieves the **data** field from the specified window. It returns NIL if no such context exists on the window. It also locks the context object so that no other calls on **Acquire** or **Destroy** with the same **type** and **window** will complete until the context is freed by a call on **Release**.

Release: PROCEDURE **[type: Type, window: Window.Handle];**

The procedure **Release** releases the lock on the specified context object for **window** that was locked by the call on **Acquire**. If the specified context cannot be found or if it is not locked, **Release** is a no-op.

21.2.4 Errors

ErrorCode: TYPE = **{duplicateType, windowIsNIL, tooManyTypes, other};**

duplicateType	is raised by **Create** if a context of the given type already exists on the window passed as an argument.
windowIsNIL	is raised if the client has passed in a NIL window.
tooManyTypes	is raised if **UniqueType** has been called too many times.

Error: ERROR **[code: ErrorCode];**

Error is the only error raised by any of the **Context** procedures.

21.3 Usage/Examples

Acquire and **Release** can be used in much the same way as a Mesa **MONITOR** (See the *Mesa Language Manual*: 610E00150). It is important that the client call **Release** for every context that has been obtained by **Acquire**; this is not done automatically. The cost of doing an **Acquire** is barely more than entering a **MONITOR** and doing a **Find**. Using this technique allows the client to monitor its data rather than its code.

If several tools must share global data, it is possible to place a context on **Window.rootWindow** that is never destroyed, even when the bitmap is turned off. To share a **Type** without having to **EXPORT** a variable, use one in the range **(lastAllocated..last]**. Contact the support organization to have one allocated to you.

21.3.1 Example

```
myContextType: Context.Type ← Context.UniqueType[];

MyContext: TYPE = LONG POINTER TO MyContextObject;

MyContextObject: TYPE = RECORD [...];

sysZ: UNCOUNTED ZONE ← Heap.systemZone;

MakeShellAndBodyWindow: PROCEDURE = {
  myContext: MyContext ← sysZ.NEW [MyContextObject ← [
  -- initialize fields of MyContextObject --] ];
  -- Note: If some field of MyContextObject were a pointer to some more allocated
  storage, then the Context.SimpleDestroyProc would not be used. A client-supplied
  DestroyProcType that freed both MyContextObject and the storage pointed to by
  MyContextObject would have to be provided.
  ...
  shell: StarWindowShell.Create [...];
  body: StarWindowShell.CreateBody [sws: shell,
    repaintProc: MyRepaint,
    bodyNotifyProc: MyNotify];
  Context.Create [type: myContextType,
    data: myContext,
    proc: Context.SimpleDestroyProc,
    window: body];
  ...
  };
```

```
MyRepaint: PROCEDURE [window: Window.Handle] = {
  myContext: MyContext ← FindContext [window];
  ...
  };

MyNotify: TIP.NotifyProc = {
  myContext: MyContext ← FindContext [window];
  ...
  };

FindContext: PROCEDURE [window: Window.Handle]
  RETURNS [myContext: MyContext] = {
  myContext ← Context.Find [myContextType, window];
  IF myContext = NIL THEN ERROR;
  };
```

21.4 Index of Interface Items

22 Cursor

22.1 Overview

The **Cursor** interface provides a procedural interface to the hardware mechanism that implements the cursor on the screen. This interface defines several cursor shapes as well as operations for client-defined cursors. Because there is a single global cursor, it should be manipulated only through this interface and only from the notifier process.

The major data structure defined in this interface is the **Object**, which defines not only the array of bits that represents the picture of the cursor but also its hot spot. The hot spot of a cursor consists of the coordinates within the 16-by-16 array that indicate the screen position pointed to by the mouse. The hardware position of the cursor is always in the upper-left corner of the bit array. For many cursor shapes, this position is not where the cursor points. For example, the **pointRight** cursor shape is a right-pointing arrow whose hot spot is at the tip of the arrow.

There can be up to 256 different cursors, limited by the size of the **Type** enumeration. The first several types are system-defined. Clients may call **UniqueType** to allocate an unused type for their own use.

This interface is typically used to change the cursor either by calling **Set** to set it to one of the system-defined cursors or by calling **Store**. To restore the cursor, save it into an **Object** by calling **Fetch** before it is changed.

22.2 Interface Items

22.2.1 Major Data Structures

```
Handle: TYPE = LONG POINTER TO Object;

Object: TYPE = RECORD [info: Info, array: UserTerminal.CursorArray];

Info: TYPE = RECORD [type: Type, hotX: [0..16), hotY: [0..16)];
```

Type: TYPE = MACHINE DEPENDENT{
blank(0), bullseye(1), confirm(2), ftpBoxes(3), hourGlass(4), lib(5), menu(6), mouseRed(7), pointDown(8), pointLeft(9), pointRight(10), pointUp(11), questionMark(12), scrollDown(13), scrollLeft(14), scrollLeftRight(15), scrollRight(16), scrollUp(17), scrollUpDown(18), textPointer(19), groundedText(20), move(21), copy(22), sameAs(23), adjust(24), row(25), column(26), last(377B)};

Object defines the type and hot spot of the cursor as well as the 16-by-16 array of bits that represent the cursor's picture.

Info contains the type and the hot spot of a cursor.

Defined: TYPE = Type[blank..column];

Defined is the subrange of **Type** that contains the system-defined cursors.

22.2.2 Setting the Cursor Picture

Set: PROCEDURE [type: Defined];

Set sets the displayed cursor to be one of the system-defined cursors.

Store: PROCEDURE [h: Handle];

Store sets the displayed cursor to the cursor described by **h**.

StoreCharacter: PROCEDURE [c: XChar.Character];

StoreCharacter stores the system font picture of character **c** into the cursor. The **info** is set to [type: column.SUCC, hotX: 8, hotY: 8].

StoreNumber: PROCEDURE [n: CARDINAL];

StoreNumber sets the cursor picture to be the number **n** MOD 100. If **n** is less than 10, the single digit is centered in the cursor. The **info** is set to [type: column.SUCC.SUCC, hotX: 8, hotY: 8].

22.2.3 Getting Cursor Information

Fetch: PROCEDURE [h: Handle];

Fetch copies the current cursor object into the object pointed to by **h**.

GetInfo: PROCEDURE RETURNS [info: Info];

GetInfo returns the hot spot and type of the current cursor.

FetchFromType: PROCEDURE [h: Handle, type: Defined];

FetchFromType copies the system-defined cursor object corresponding to **type** into the object pointed to by **h**.

22.2.4 Miscellaneous Operations

```
MoveIntoWindow: PROCEDURE [
   window: Window.Handle, place: Window.Place];
```

MoveIntoWindow moves the cursor to the window-relative **place** in **window**.

```
Swap: PROCEDURE [old, new: Handle];
```

Swap places the displayed cursor object in **old**↑ and **Stores** the **new**. It is equivalent to **Fetch[old]; Store[new]**.

22.2.5 Client-Defined Cursors

```
UniqueType: PROCEDURE RETURNS [Type];
```

UniqueType lets clients assign a unique type to their defined cursors. It returns a **Type** that is different from all predefined types and from any that have previously been returned by **UniqueType**. The value is only valid during the current boot session.

22.2.6 Cursor Picture Manipulation

```
Invert: PROCEDURE RETURNS [BOOLEAN];
```

Invert inverts each bit of the cursor picture and inverts the positive/negative state of the picture. It returns TRUE if the new state of the cursor is positive.

```
MakeNegative: PROCEDURE;
```

MakeNegative is equivalent to **MakePositive** followed by **Invert**. It sets the positive/negative state of the cursor to negative.

```
MakePositive: PROCEDURE;
```

MakePositive sets the positive/negative state of the cursor to positive. The state is set to positive whenever **Set** or **Store** is invoked.

22.3 Usage/Examples

The following example shows a client setting the cursor to an hourglass while performing some time-consuming action. It first saves the current cursor and restores it when it is done, if the action did not change the cursor. If the client knew what the cursor should be, the cursor would not have to be saved but could be unconditionally set.

```
savedCursor: Cursor.Object;

Cursor.Fetch[@savedCursor];
Cursor.Set[hourGlass]
-- Do action --
IF Cursor.GetInfo[].type = hourGlass THEN Cursor.Store[@savedCursor];
```

StoreCharacter is typically used to put small pictures in the cursor by using characters obtained from **SimpleTextFont.AddClientDefinedCharacter**.

22.4 Interface Item Index

Item	Page
Defined: TYPE	2
Fetch: PROCEDURE	2
FetchFromType: PROCEDURE	2
GetInfo: PROCEDURE	2
Handle: TYPE	1
Info: TYPE	1
Invert: PROCEDURE	3
MoveIntoWindow: PROCEDURE	3
MakeNegative: PROCEDURE	3
MakePositive: PROCEDURE	3
Object: TYPE	1
Set: PROCEDURE	2
Store: PROCEDURE	2
StoreCharacter: PROCEDURE	2
StoreNumber: PROCEDURE	2
Swap: PROCEDURE	3
Type: TYPE	2
UniqueType: PROCEDURE	3

23

Directory

23.1 Overview

Directory allows for clients to add dividers to the directory icon. **Directory** maintains a directory divider containing three top-level dividers: the workstation divider, containing those objects that exist on a per-workstation basis; the user divider, containing those objects that exist on a per-user or per-desktop basis; and the network divider, containing those objects that exist in the internet. (See the **Divider** and **CHDivider** interfaces for more information about dividers.)

23.1.1 Predefined Divider Structure

Directory automatically creates a top-level divider that backs the directory icon. To this divider it adds the workstation divider, the user divider, and the network divider. It adds three entries to the workstation divider: the prototype folder, the office aids divider, and the local devices divider. The user divider is emptied at each logout. Clients of the user divider should add their entries at each logon. **Directory** also automatically adds the organization divider to the network divider and the domain divider to the organization divider. Clients can add entries to the domain divider (see Figure 23.1). (See the **Prototype** interface for details of how to add prototype icons to the prototype folder and the **Divider** interface for details of how to add entries to the office aids, local devices, and user dividers.)

23.2 Interface Items

23.2.1 Adding Items to a Predefined Divider

```
DividerType: TYPE = {top, ws, user, domain, localDevices, officeAids};
```

A parameter of type **DividerType** is passed to **AddDividerEntry** to specify one of the predefined dividers. A value of **top** specifies adding a new top-level divider.

```
AddDividerEntry: PROCEDURE [
    divider: DividerType,
    type: NSFile.Type,
label: XString.Reader,
data: LONG POINTER ← NIL,
convertProc: Divider.ConvertProc ← NIL,
genericProc: Divider.GenericProc ← NIL];
```

AddDividerEntry adds an entry to the divider specified by **divider**. If **divider** is equal to **top**, a new top-level divider is added. **type** specifies the **NSFile.Type** of the entry. It is used to obtain the **Containee.Implementation** for the entry. **label** is used to label the entry when it appears in the divider's container window. The **XString.Reader** bytes are copied. **data** is an optional data pointer to be supplied in subsequent calls to the **GenericProc** and the **ConvertProc**. **convertProc** is a **Divider.ConvertProc** for the entry, and **genericProc** is a **Divider.GenericProc** for the entry. (See the **Divider** interface for details.) Fine point: The predefined dividers are actually implemented by using the **Divider** interface. **AddDividerEntry** is actually the same as **Divider.AddEntry**, with the **handle** arguement replaced by a **Directory.DividerType**.

23.2.2 GetDividerHandle

```
GetDividerHandle: PROCEDURE [divider: DividerType] RETURNS [handle: Divider.Handle];
```

GetDividerHandle returns the **Divider.handle** for the predefined divider specified by **divider**. Clients can use this handle to manipulate the predefined divider with the **Divider** interface. (See the **Divider** chapter for more information.)

23.3 Usage/Examples

See the **Divider** and **CHDivider** interfaces for examples of how to add entries to the directory. The **Divider** interface also shows the implementation of **AddDividerEntry**.

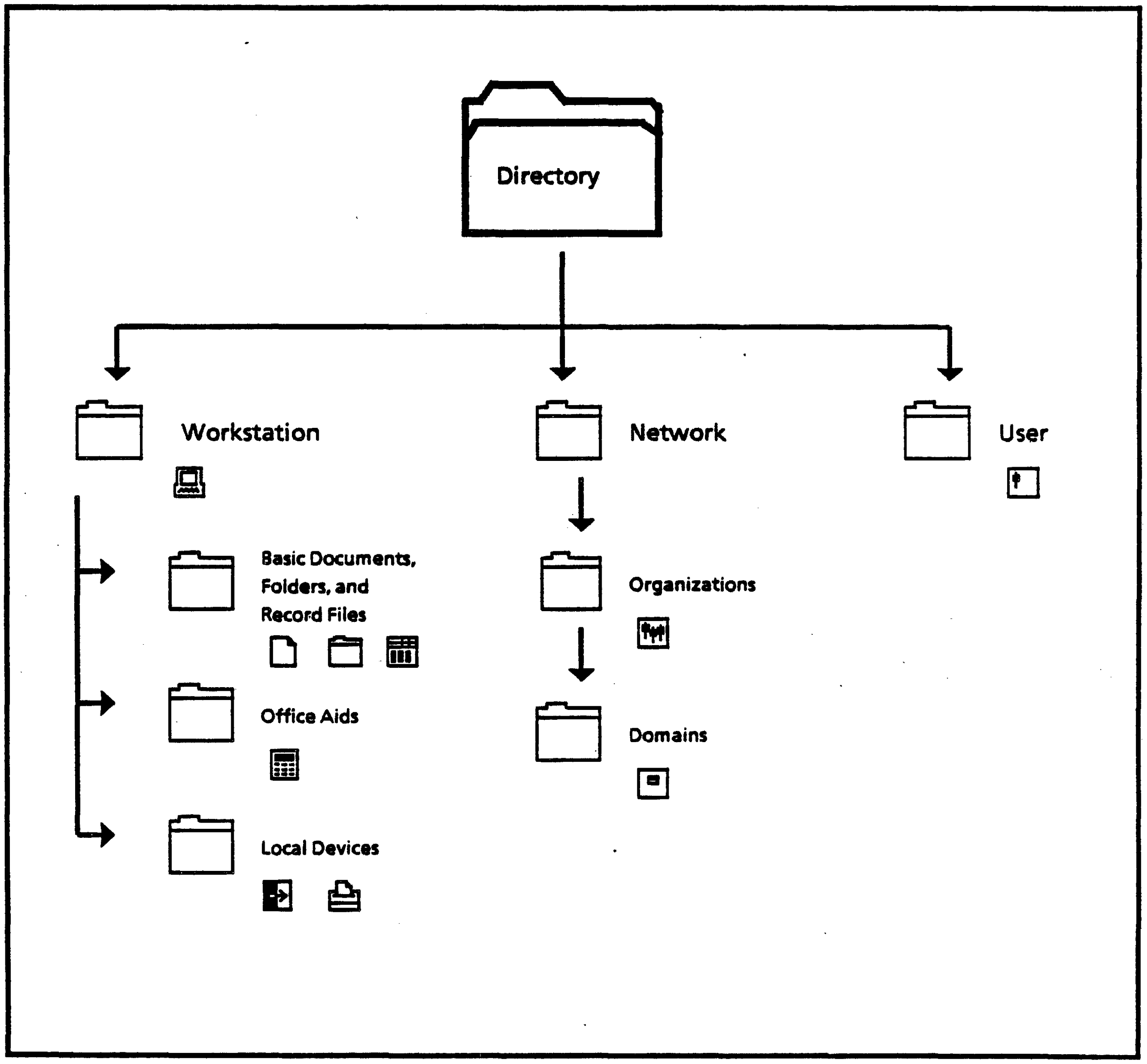


Figure 23.1 Predefined Divider Structure

23.4 Index of Interface Items

Item	Page
AddDividerEntry: PROCEDURE	1
GetDividerHandle: PROCEDURE	2
DividerType: TYPE	1

24

Display

24.1 Overview

The **Display** interface provides elementary routines for painting into windows on the display screen. Procedures are provided for painting points; lines; bitmaps; repeating patterns; boxes filled with black, gray, white, or small patterns; circles; circular arcs; ellipses; conics; as well as for painting a brush as it moves along an arbitrary trajectory. Another procedure allows shifting the current content of a window. Procedures for painting text are available in the **SimpleTextDisplay** interface.

The **Window** interface supplies facilities for managing windows. The introduction section of the **Window** chapter describes the window coordinate system and the process of painting into a window. The reader should be familiar with that material.

As described in the **Window** chapter, the display background color, which is represented by a pixel value of zero, is commonly called *white*, and a value of one, called *black*. Note however, that the display hardware can also render the picture using zero for black and one for white. *Clearing* or *erasing* an area of the screen means setting all of its pixels to zero, or white.

The **Display** interface currently contains procedures that apply to text--namely **Block**, **MeasureBlock**, **ResolveBlock**, **Character**, **Text**, and **TextInline**. *They are not supported.* The **SimpleTextDisplay** interface provides text painting operations.

As described in the **Window** chapter, the standard way for a client to paint into its window is to update its data structures, invalidate the portion of its window that needs to be painted, and then call a **Window.Validate** routine. **Window** responds by calling back into the client's display procedure to do the painting. Nonstandard ways of painting are discussed in the Usage/Examples section of this chapter.

24.2 Interface Items

24.2.1 Painting Filled Boxes, Horizontal Lines, and Vertical Lines

```
Handle: TYPE = Window.Handle;
```

```
Black: PROCEDURE [window: Window.Handle, box: Window.Box];

Invert: PROCEDURE [window: Window.Handle, box: Window.Box];

White: PROCEDURE [window: Window.Handle, box: Window.Box];
```

Black and **White** paint black and white boxes. **Invert** changes all black pixels to white and all white pixels to black in the box. These procedures perform their operation on the specified **box** in **window**. Horizontal and vertical black lines can be painted by using **Black** with a box that is one pixel wide or tall.

Display.Handle is provided for backward compatibility.

24.2.2 Painting Bitmaps and Gray Bricks

The procedures in this section allow the client to paint bitmaps and gray bricks into a window. Bitmaps and gray bricks are described in the *Mesa Processor Principles of Operation*.

The first items below define some convenience types and constants that are used with bitmaps and painting.

```
BitAddress: TYPE = Environment.BitAddress;

DstFunc: TYPE = BitBlt.DstFunc;

BitBltFlags: TYPE = BitBlt.BitBltFlags;
```

A **BitBlt.BitBltFlags** is an argument of the **Bitmap** and **Trajectory** operations. These flags control how source pixels and existing display pixels are combined to produce the final display pixels. The flag constants defined below cover most of the common cases. **BitBlt.BitBltFlags** are described in detail in the *Mesa Processor Principles of Operation.*

```
replaceFlags: BitBltFlags = [
    direction: forward, disjoint: TRUE, disjointItems: TRUE, gray: FALSE,
    srcFunc: null, dstFunc: null, reserved: 0];
```

replaceFlags paints opaque black and opaque white from a bitmap. Source pixels from the bitmap overwrite the previous display pixels.

```
textFlags, paintFlags: BitBltFlags = [
    direction: forward, disjoint: TRUE, disjointItems: FALSE, gray: FALSE,
    srcFunc: null, dstFunc: or, reserved: 0];
```

textFlags and its synonym **paintFlags** paint opaque black and transparent white from a bitmap source. Black source pixels cause black display pixels. White source pixels leave display pixels unchanged.

```
xorFlags: BitBltFlags = [
    direction: forward, disjoint: TRUE, disjointItems: FALSE, gray: FALSE,
    srcFunc: null, dstFunc: xor, reserved: 0];
```

xorFlags is used with a source bitmap to selectively *video invert* existing display pixels. Video inverting is the process of changing white to black and black to white. Black source pixels invert the existing display pixels. White source pixels leave display pixels unchanged.

```
paintGrayFlags, bitFlags: BitBltFlags = [
    direction: forward, disjoint: TRUE, disjointItems: TRUE, gray: TRUE,
    srcFunc: null, dstFunc: or, reserved: 0];
```

paintGrayFlags paints opaque black and transparent white from a gray brick source. Black source pixels cause black display pixels. White source pixels leave display pixels unchanged.

```
replaceGrayFlags, boxFlags: BitBltFlags = [
    direction: forward, disjoint: TRUE, disjointItems: TRUE, gray: TRUE,
    srcFunc: null, dstFunc: null, reserved: 0];
```

replaceGrayFlags paints opaque black and opaque white from a gray brick source. Source pixels overwrite the previous display pixels.

```
xorGrayFlags, xorBoxFlags: BitBltFlags = [
    direction: forward, disjoint: TRUE, disjointItems: TRUE, gray: TRUE,
    srcFunc: null, dstFunc: xor, reserved: 0];
```

xorGrayFlags is used with a source gray brick to selectively *video invert* existing display pixels. Black source pixels invert the existing display pixels. White source pixels leave display pixels unchanged.

```
eraseFlags: BitBltFlags = [
    direction: forward, disjoint: FALSE, disjointItems: FALSE, gray: FALSE,
    srcFunc: complement, dstFunc: and, reserved: 0];
```

eraseFlags erases objects. Previous display pixels are overwritten.

```
Bitmap: PROCEDURE [
    window: Window.Handle, box: Window.Box, address: Environment.BitAddress,
    bitmapBitWidth: CARDINAL, flags: BitBlt.BitBltFlags ← paintFlags];
```

Bitmap paints the bitmap described by **address** and **bitmapBitWidth** into **box** in **window**, using **flags** to control the interaction with pixels already being displayed. **Bitmap** may be used to display a gray pattern that is not aligned relative to the window origin. **box.dims.w** must be less than or equal to **bitmapBitWidth**; this is not checked. **flags.gray** is ignored.

```
BitAddressFromPlace: PROCEDURE [
    base: Environment.BitAddress, x, y: NATURAL, raster: CARDINAL]
    RETURNS [Environment.BitAddress];
```

BitAddressFromPlace returns the **Environment.BitAddress** of the pixel at coordinates **x** and **y** in the bitmap described by **base**. **raster** is the number of pixels per line in the bitmap. This procedure is useful for calculating the **address** parameter of **Bitmap**.

```
Brick: TYPE = LONG DESCRIPTOR FOR ARRAY OF CARDINAL;
```

Bricks are used by **Gray** and **Trajectory** to describe a repeating pattern to fill an area. The maximum size of a **Brick** is 16 words; each word is one row of the pattern.

```
fiftyPercent: Brick;
```

fiftyPercent is a brick containing a 50% gray pattern.

```
Gray: PROCEDURE [
   window: Window.Handle, box: Window.Box, gray: Brick ← fiftyPercent,
   dstFunc: BitBlt.DstFunc ← null];
```

Gray uses the source **gray** brick to completely fill **box** in **window**. If the content of the brick to be displayed is not aligned with the window origin, use **Bitmap** instead. The table below describes the effect of **dstFunc**.

dstFunc	resulting display pixels
null	Source pixels overwrite display pixels.
or	Black source pixels cause black display pixels. White source pixels leave display pixels unchanged.
xor	Black source pixels cause the existing display pixels to be inverted. White source pixels leave display pixels unchanged.
and	Black source pixels cause black display pixels wherever the display pixels are already black. All other display pixels will be made white.

24.2.3 Painting Points, Slanted Lines, and Curved Lines

The procedures below paint points, oblique straight lines, and circular arcs and conics.

```
Point: PROCEDURE [window: Window.Handle, point: Window.Place];
```

Point makes the single pixel at **point** in **window** black.

```
LineStyle: TYPE = LONG POINTER TO LineStyleObject;

LineStyleObject: TYPE = RECORD [
  widths: ARRAY [0..DashCnt) OF CARDINAL,
  thickness: CARDINAL];

DashCnt: CARDINAL = 6;
```

LineStyle describes the style of lines for the **Line**, **Circle**, **Ellipse**, **Arc**, and **Conic** operations. **thickness** defines the width of the line in pixels. **widths** defines the dash structure. Each pair of elements is the number of pixels of black followed by the number of pixels of white. For example **[widths: [4,2,0,0,0,0], thickness: 2]** defines the style for a dashed line two pixels thick, where the dashes are four pixels on and two off.

```
Line: PROCEDURE [
  window: Window.Handle, start, stop: Window.Place, lineStyle: LineStyle ←NIL,
  bounds: Window.BoxHandle ←NIL];
```

Line paints a line from **start** to **stop** in **window**. If **bounds** # NIL, the line is clipped to the box **bounds**. If **lineStyle** is defaulted, the line is solid and is a single pixel wide.

```
Circle: PROCEDURE [
    window: Window.Handle, place: Window.Place, radius: INTEGER,
    lineStyle: LineStyle ←NIL, bounds: Window.BoxHandle ←NIL];
```

Circle paints a circle centered at **place** in **window**, with the given **radius**. If **bounds** # NIL, the circle is clipped to the box **bounds**. If **lineStyle** is defaulted, the circle is solid and is a single pixel wide.

```
Ellipse: PROCEDURE [
    window: Window.Handle, center: Window.Place, xRadius, yRadius: INTEGER,
    lineStyle: LineStyle ←NIL, bounds: Window.BoxHandle ←NIL];
```

Ellipse paints an ellipse with axes centered at **center** with an x radius of **xRadius** and a y radius of **yRadius** in **window**. The axes of the ellipse are parallel to the x-y coordinate system. Ellipses with oblique axes may be displayed by using **Conic**. If **bounds** # NIL, the ellipse is clipped to the box **bounds**. If **lineStyle** is defaulted, the ellipse is solid and is a single pixel wide.

```
Arc: PROCEDURE [
    window: Window.Handle, place: Window.Place, radius: INTEGER,
    startSector, stopSector: CARDINAL, start, stop: Window.Place,
    lineStyle: LineStyle ←NIL, bounds: Window.BoxHandle ←NIL];
```

Arc paints a portion of a circular arc centered at **place** in **window**, with the given **radius**. The arc goes from the angle defined by **start** in the **startSector** to **stop** in the **stopSector**. Sectors are simply octants numbered from 1 to 8, with northeast being 1 and increasing clockwise. If **bounds** # NIL, the arc is clipped to the box **bounds**. If **lineStyle** is defaulted, the arc is solid and is a single pixel wide.

```
Conic: PROCEDURE [
    window: Window.Handle, a, b, c, d, e, errorTerm: LONG INTEGER,
    start, stop, errorRef: Window.Place,
    sharpCornered, unboundedStart, unboundedStop: BOOLEAN,
    lineStyle: LineStyle ←NIL, bounds: Window.BoxHandle ←NIL];
```

Conic paints the portion of the curve of the equation $ax^2 + by^2 + cxy + dx + ey + f = 0$ in **window** from **start** to **stop**. Instead of passing in the last coefficient f, this procedure takes the **errorTerm** resulting from substituting **start** into the equation. If the conic contains points whose radius of curvature is less than or equal to two pixels, it must be displayed by using multiple calls with **sharpCornered** set to TRUE; otherwise ,**sharpCornered** should be FALSE. These "sharp-cornered" conics must be broken up into segments where the corners become a new segment's start and stop points. For example, a very long skinny ellipse must be displayed in two pieces. **errorRef**, **unboundedStart**, and **unboundedStop** are ignored. If **bounds** # NIL, the conic is clipped to the box **bounds**. If **lineStyle** is defaulted, the conic is solid and is a single pixel wide.

24.2.4 Painting Parallelograms and Trapezoids

These types and procedures are used to paint parallelograms and trapezoids:

```
FixdPtNum: TYPE = MACHINE DEPENDENT RECORD [
    SELECT OVERLAID * FROM
        wholeThing = > [li: LONG INTEGER],
        parts = > [frac: CARDINAL, int: INTEGER],
        ENDCASE];
```

A **FixdPtNum** is a fixed-point integer with 16 bits of fraction and 16 bits of integer part. These numbers can be added and subtracted in a straightforward manner, while division and multiplication are more difficult. By using the overlaid record, the fraction and integer part may be obtained without shifting or dividing. **FixdPtNum** can express all practical slopes with only small errors.

```
Interpolator: TYPE = RECORD [
    val, dVal: FixdPtNum];
```

Interpolator is used to define parallelograms and trapezoids. The **dVal** term is the derivative with respect to y; for example, **x.dVal** is dx/dy.

```
BlackParallelogram: PROC [
    window: Handle, p: Parallelogram, dstFunc: DstFunc ← null];
```

```
Parallelogram: TYPE = RECORD [
    x: Interpolator, y: INTEGER, -- upper left
    w: NATURAL, -- across top, must be positive
    h: NATURAL];
```

BlackParallelogram paints the parallelogram defined by **p** in **window**. **dstFunc** acts as in the procedure **Gray**. The parallelogram is defined as below with the slope of the parallelogram being **p.x.dVal**. In Figure 24.1 the slope is two fifths. **BlackParallelogram**

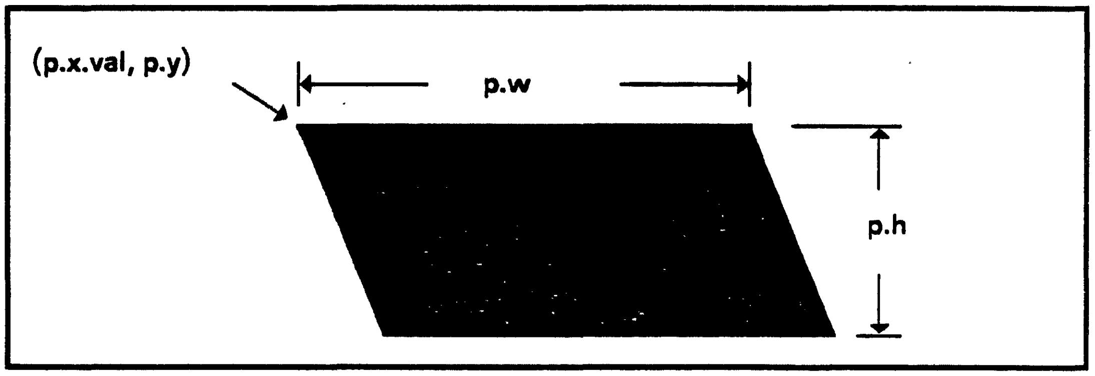


Figure 24.1 Parallelogram definition

optimizes a common case (such as diagonal lines) and runs about twice as fast as

GrayTrapezoid by avoiding the second interpolation, the noninteger width, and the gray alignment calulations:

```
GrayTrapezoid: PROC [
    window: Handle, t: Trapezoid, gray: Brick ← fiftyPercent, dstFunc: DstFunc ← null];

Trapezoid: TYPE = RECORD [
    x: Interpolator, y: INTEGER, -- upper left
    w: Interpolator, -- across top; must be positive
    h: NATURAL];
```

GrayTrapezoid paints the trapezoid defined by **t** in **window**. **gray** and **dstFunc** act as in the procedure **Gray**. The trapezoid is defined in Figure 24.2 with the slope of the left side of the trapezoid being **t.x.dVal** and the slope of the right side of the trapezoid being **t.x.dVal** minus **t.w.dVal**. In Figure 24.2, **t.x.dVal** is minus one half and **t.w.dVal** is nine tenths.

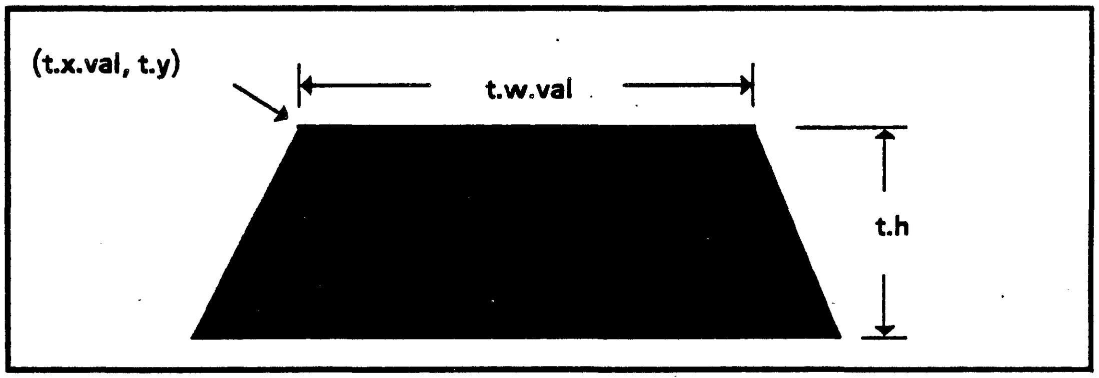


Figure 24.2 Trapezoid definition

24.2.5 Painting Along Trajectories, Shifting Window Contents

```
Shift: PROCEDURE [window: Window.Handle, box: Window.Box, newPlace: Window.Place];
```

Shift does a block move of a rectangular portion of **window**'s current content. This operation does not invoke any client display procedures. **box** describes the region of **window** to be moved to **newPlace**. If **Display** does not have the pixels for a visible area of the destination box, that area is filled with trash and marked invalid. The client should validate the window when it has finished altering the window content. **Shift** does not invalidate the areas vacated by the move; if they are repainted, the client should invalidate them. If **Shift** is executed from within a display procedure, it does not clip the region painted to window's invalid area list. Invalid area lists are explained in the **Window** chapter.

```
Trajectory: PUBLIC PROCEDURE [
   window: Window.Handle, box: Window.Box ← Window.nullBox, proc: TrajectoryProc,
   source: LONG POINTER ← NIL, bpl: CARDINAL ← 16, height: CARDINAL ← 16,
   flags: BitBlt.BitBltFlags ← bitFlags, missesChildren: BOOLEAN ← FALSE,
   brick: Brick ← NIL];

TrajectoryProc: TYPE = PROCEDURE [Handle] RETURNS [Window.Box, INTEGER];
```

Trajectory repeatedly calls **proc** and paints a brush where **proc** specifies. The brush may be either a gray **brick** or a portion of the bitmap **source**. **Trajectory** avoids much of the overhead of successive calls to the normal **Display** routines. **box** is the window region in which painting may occur. The client must not try to paint outside **box**; this is not checked. **flags** controls the type of painting performed. If **flags.gray** = TRUE, the gray **brick** is painted; otherwise, a bitmap is painted. **Trajectory** repeatedly calls **proc** for instructions. If **proc** returns a box having **dims.w** = 0 (such as **Window.nullBox**), iteration ceases and **Trajectory** returns. Otherwise **dims.w** # 0; **Trajectory** paints the brush and then loops to call **proc** again. The brush paints the returned **Box** in the window as follows. If a gray brick is being painted, the brick completely fills the returned **Box**. If a bitmap is being painted, the bitmap starts at a bit offset of <INTEGER> from **source**, is **Box.dims.h** high, and has **bpl** pixels per line. The client may wish to alter the brush content along the trajectory by having **source** be a large bitmap containing several different brush patterns and having **proc** return the bit offset and **Box.dims** of the desired portion. (**BitBlt.BitBltFlags** are described in §24.2.2.) **height** and **missesChildren** are unused. **proc** must not call any procedures in **Display** or **Window**; doing so will result in a deadlock.

24.3 Usage/Examples

24.3.1 Special Topic: Direct Painting

As described in the **Window** chapter, the standard way for a client to paint into its window is to update its data structures, invalidate the portion of its window that needs to be painted, and then call a **Window.Validate** routine. **Window** responds by calling back into the client's display procedure to do the painting.

The client may also paint directly into a window without going through **Window.Validate**. However, this direct-painting approach is subject to several pitfalls and system bugs. Clients commonly choose direct painting only when high painting performance is required, such as dynamically extending an inverted selection while tracking the mouse or implementing a blinking caret.

Pitfall 1: One consequence of doing direct painting is that *the window's display procedure must not depend on* **Window** *clearing invalid areas for it.* As described in the Window chapter, if **clearingRequired** = TRUE, **Window** guarantees that when the display procedure is called to paint the window, all of the window's pixels that should be white indeed are white. In that situation, the window might contain any combination of its previous contents and erased areas. Notice that the following sequence of events might occur: **Window** clears invalid area; then the client direct paints into some part of the invalid area; then **Window** calls the window's display procedure. In this situation, the parallel direct-paint activity has voided **Window**'s guarantee of the content of the invalid area. To handle this case, the display routine must erase or otherwise completely overpaint the invalid areas itself.

Pitfall 2: A client can get into trouble when it wishes to change the state of the backing data being displayed within a display procedure and attempts to make the change by painting from the display procedure rather than by invalidating the affected area and painting later. The display procedure's paint is clipped to its invalid area list and thus fails to achieve the desired effect. There are several ways to solve this problem:

- Do not change the backing data inside a display procedure. This approach matches nicely with the intended function of a display procedure. Do not expect a display procedure to change data—its job is to repaint.

- Have the display procedure just invalidate the areas affected by the data being changed. Because a validate is already in progress, it is not necessary to call Window.Validate. When the display procedure returns, it is called back with any new invalid areas that are waiting for it.

- Have the display procedure call Window.FreeBadPhosphorList before changing the data. This allows paint from the display procedure to affect the entire window, not just the invalid areas.

24.3.2 Example 1

The program fragments below demonstrate the use of Display in a window's display procedure.

```
-- Enumerated TYPEs for displaying the games background.
Background: TYPE = {gray, white};
background: Background ← gray;

DisplayBoardSW: PROC [window: Window.Handle] = {
    -- This is the body window's display procedure.
    vLine, hLine: Window.Box;
    left, right, top, bottom: INTEGER;

    FindBounds: PROC [window: Window.Handle, box: Window.Box] = {
        left ← MIN[left, box.place.x];
        top ← MIN[top, box.place.y];
        right ← MAX[right, box.place.x + box.dims.w];
        bottom ← MAX[bottom, box.place.y + box.dims.h]};

    -- Paint borders and background.
    Display.Black[window: window, box: boardAndBorderBox];
    PaintBackground[window: window, box: boardBox];
    vLine ← [upperLeft, [lineWidth, (boardSize - 1)*unitH + 1]];
    hLine ← [upperLeft, [(boardSize - 1)*unitW + 1, lineWidth]];
    THROUGH [firstDimboardSize] DO
        Display.Black[window, vLine];
        Display.Black[window, hLine];
        vLine.place:x ← vLine.place.x + unitW;
        hLine.place.y ← hLine.place.y + unitH;
        ENDLOOP;

    .
    .
    left ← top ← INTEGER.LAST;
    right ← bottom ← INTEGER.FIRST;
    window.EnumerateInvalidBoxes[FindBounds]
    .
    .
    };

PaintBackground: PROC [window: Window.Handle, box: Window.Box] = {
    SELECT background FROM
        gray => Display.Gray[window, box];
        white => Display.White[window, box];
        ENDCASE
    };

PaintStone: PUBLIC PROC [who: BlackWhite, u, v: Dim, play: CARDINAL] = {
    center: Window.Place;
    stoneBox: Window.Box;
    numStr: STRING ← [3];

    IF -ValidCoords[u, v] THEN RETURN;
    center ← BoardToPlace[u, v];
    stoneBox ← [
    place: [center.x - stoneRadius, center.y - stoneRadius],
    dims: [stoneSize, stoneSize]];

    -- paint a bitmap that represents game pieces.
    Display.Bitmap[
        window: boardSW, box: stoneBox, address: outerStone,
        bitmapBitWidth: stoneBpl, flags: Display.paintFlags];
    IF who = white THEN
    Display.Bitmap[
        window: boardSW, box: stoneBox, address: innerStone,
        bitmapBitWidth: stoneBpl, flags: eraseFlags];
    .
    .
    };

CreateGoSWS: PUBLIC PROCEDURE [
    reference: NSFile.ReferenceRecord, name: Environment.Block ]
    RETURNS [StarWindowShell.Handle] = {
    -- This procedure is invoked via a system menu.
    sz: StarWindowShell.Handle;
    .
    .
    StarWindowShell.SetPreferredDims [ sz, [592, 661] ];
    -- The display procedure is set here.
    boardSW ← StarWindowShell.CreateBody [
        sws: sz,
        repaintProc: DisplayBoardSW,
        bodyNotifyProc: TIPMe ];
    .
    .
    };
```

24.4 Index of Interface Items

Item	Page
Arc: PROCEDURE	5
BitAddress: TYPE	2
BitAddressFromPlace: PROCEDURE	3
BitBltFlags: TYPE	2
bitFlags: BitBlt.BitBltFlags	3
Bitmap: PROCEDURE	3
Black: PROCEDURE	2
BlackParallelogram: PROCEDURE	6
boxFlags: BitBlt.BitBltFlags	3
Brick: TYPE	4
Circle: PROCEDURE	5
Conic: PROCEDURE	5
DashCnt: PROCEDURE	4
DstFunc: TYPE	2
Ellipse: PROCEDURE	5
eraseFlags: BitBlt.BitBltFlags	3
fiftyPercent: Brick	4
FixdPtNum: TYPE	6
Gray: PROCEDURE	4
GrayTrapezoid: PROCEDURE	7
Handle: TYPE	1
Interpolater: TYPE	6
Invert: PROCEDURE	2
Line: PROCEDURE	5
LineStyle: TYPE	4
LineStyleObject: TYPE	4
paintBitFlags:BitBlt.BitBltFlags	3
paintFlags: BitBlt.BitBltFlags	2
paintGrayFlags: BitBlt.BitBltFlags	3
Parallelogram: TYPE	6
Point: PROCEDURE	4
replaceboxFlags: BitBlt.BitBltFlags	3
replaceFlags: BitBlt.BitBltFlags	2
replaceGrayFlags: BitBlt.BitBltFlags	3
Shift: PROCEDURE	7
textFlags: BitBlt.BitBltFlags	2
Trajectory: PROCEDURE	8
TrajectoryProc: TYPE	8
Trapezoid: TYPE	7
White: PROCEDURE	2
xorBoxFlags: BitBlt.BitBltFlags	3
xorFlags: BitBlt.BitBltFlags	2
xorGrayFlags: BitBlt.BitBltFlags	3

25

Divider

25.1 Overview

Divider maintains a table of entries in memory, each representing an icon. The entries may or may not be backed by files. Divider does not operate on these entries directly; it uses a Divider.ConvertProc and a Divider.GenericProc associated with each entry.

Also associated with each entry is an NSFile.Type used to identify the entry's Containee.Implementation, a label, and a pointer to instance-specific data for the entry.

Associated with each divider when it is created is an NSFile.Type. Divider automatically sets a Containee.Implementation for this file type that supports converting the divider to a file and opening the divider as a container window displaying the entries.

Also associated with each divider is a CH.Pattern specifying a clearinghouse domain and organization. It is inherited from a parent divider and is passed to all entries through the Divider.ConvertProc and the Divider.GenericProc associated with each entry. When the divider is converted to a file, the pattern is automatically encoded in an attribute of the file.

25.2 Interface Items

25.2.1 Creating and Destroying

```
Handle: TYPE = LONG POINTER TO Object;

Object: TYPE;

Create: PROCEDURE [
    type: NSFile.Type,
    name: XString.Reader,
    initialSize: CARDINAL ← Divider.defaultInitialSize,
    increment: CARDINAL ← Divider.defaultIncrement,
    zone: UNCOUNTED ZONE ← NIL]
    RETURNS [handle: Handle];
```

Create creates a divider. **type** specifies the **NSFile.Type** the divider has if it is converted to a file. A **Containee.Implementation** is automatically set for this type. **name** specifies the name of the divider. It appears in the window header when the divider is opened, and it is the name of the file if the divider is converted to a file. The **XString.Reader** bytes are copied. The divider is created with a table large enough to hold **initialSize** entries. If an entry is added when the table is full, the table grows by **increment** entries. Storage for the divider is allocated from **zone**. If **zone** is defaulted, storage is allocated from a heap maintained by **Divider**.

```
Destroy: PROCEDURE [handle: Handle];
```

This releases all storage associated with the given **divider**. **handle** is no longer valid when this procedure returns.

25.2.2 ConvertProc and GenericProc

```
ConvertProc: TYPE = PROCEDURE [
    data: LONG POINTER,
    pattern: CH.Pattern,
    target: Selection.Target,
    zone: UNCOUNTED ZONE,
    info: Selection.ConversionInfo ← [convert[]]]
    RETURNS [value: Selection.Value];
```

A **ConvertProc** is the same as a **Selection.ConvertProc** except that it has the extra argument, **pattern**, that specifies a clearinghouse domain and organization. (See the **Selection** interface for the definition of the other arguments.) Whenever the divider is requested to convert one of its entries, it calls the **ConvertProc** associated with an entry, with **pattern** set to the domain and organization associated with the divider,

```
GenericProc: TYPE = PROCEDURE [
    atom: Atom.ATOM,
    data: LONG POINTER,
    pattern: CH.Pattern,
    changeProc: Containee.ChangeProc ← NIL,
    changeProcData: LONG POINTER ← NIL]
    RETURNS [LONG UNSPECIFIED];
```

A **GenericProc** is the same as a **Containee.GenericProc** except that it has the extra argument, **pattern**, that specifies a clearinghouse domain and organization. (See the **Containee** interface for the definition of the other arguments.) Whenever the divider is requested to operate on one of its entries, it calls the **GenericProc** associated with an entry, with **pattern** set to the domain and organization associated with the divider.

```
DividerConvertProc: ConvertProc;
```

```
DividerGenericProc: GenericProc;
```

These procedures may be associated with entries that themselves are dividers. In this case the **Handle** associated with the divider should be provided as the instance-specific data handle. See below for an example of a divider contained in another divider.

25.2.3 Adding and Finding Entries

```
AddEntry: PROCEDURE [
    handle: Handle,
    type: NSFile.Type,
    label: XString.Reader,
    data: LONG POINTER ← NIL,
    convertProc: ConvertProc ← NIL,
    genericProc: GenericProc ← NIL];
```

AddEntry adds an entry to the divider specified by **handle**. **type** obtains the **Containee.Implementation** for the entry. **label** is used to label the entry in the divider's container window. The **XString.Reader** bytes are copied. **data** is item-specific data for the entry that is passed to the **ConvertProc** and **GenericProc** associated with the entry. If **convertProc** or **genericProc** is defaulted, the divider uses the corresponding procedure in the entry's **Containee.Implementation**.

```
FindEntry: PROCEDURE [handle: Handle, type: NSFile.Type,
    label: XString.Reader]
    RETURNS [found: BOOLEAN, entryData: LONG POINTER];
```

FindEntry finds the entry in the divider **handle** with the specified **type** and **label**. **found** indicates whether the item was in the divider. **entryData** is the data associated with the entry, if it was found. **FindEntry** is defined in DividerExtra.mesa.

```
FindOrAddEntry: PROCEDURE [handle: Handle, type: NSFile.Type,
    label: XString.Reader, data: LONG POINTER ← NIL,
    convertProc: ConvertProc ← NIL,
    genericProc: GenericProc ← NIL]
    RETURNS [found: BOOLEAN, entryData: LONG POINTER];
```

FindOrAddEntry finds the entry in the divider **handle** with the specified **type** and **label**, and adds an entry if it was not found. **found** indicates whether the item was in the divider. **entryData** is the data associated with the entry, if it was found. **FindOrAddEntry** is defined in DividerExtra.mesa.

25.3 Usage/Examples

25.3.1 Fragment from DirectoryImpl.mesa

This fragment is from **DirectoryImpl.mesa**, which implements the **Directory** interface. It shows the implementation of **Directory.AddDividerEntry** and the mainline code to create the top-level directory dividers. See the **CHDivider** interface for more examples.

```
-- File types for the directory implementation --
directory: StarFileTypes.FileType = ...;
folder: StarFileTypes.FileType = ...;
workstation: StarFileTypes.FileType = ...;
user: StarFileTypes.FileType = ...;
domain: StarFileTypes.FileType = ...;
```

```
-- The reference for the prototype folder --
prototypeReference: NSFile.Reference ← ...;

-- Handles for the top-level dividers --
dividers: ARRAY Directory.DividerType OF Divider.Handle ← ALL [NIL];

AddDividerEntry: PUBLIC PROCEDURE [
    divider: Directory.DividerType,
    type: NSFile.Type,
    label: XString.Reader,
    data: LONG POINTER ← NIL,
    convertProc: Divider.ConvertProc ← NIL,
    genericProc: Divider.GenericProc ← NIL] =
BEGIN
    Divider.AddEntry [
        handle: dividers[divider],
        type: type,
        label: label,
        data: data,
        convertProc: convertProc,
        genericProc: genericProc];
END;

-- Create the top-level dividers (top will back the directory icon) --
dividers[top] ← Divider.Create [directory, stringDirectory];
dividers[ws] ← Divider.Create [workstation, stringWorkstation];
dividers[user] ← Divider.Create [user, stringUser];

-- Insert the workstation divider into the directory --
Directory.AddDividerEntry [
    divider: top,
    type: workstation,
    label: stringWorkstation,
    data: dividers[ws],
    convertProc: Divider.DividerConvertProc,
    GenericProc: Divider.DividerGenericProc];


-- Insert the user divider into the directory --
Directory.AddDividerEntry [
    divider: top,
    type: user,
    label: stringUser,
    data: dividers[user],
    convertProc: Divider.DividerConvertProc,
    genericProc: Divider.DividerGenericProc];


-- Insert the prototype folder into the workstation divider --
-- (Note: this is an actual file that will use the folder implementation) --
Directory.AddDividerEntry [
    divider: ws,
type: folder,
label: stringPrototypes,
data: @prototypeReference];
```

25.4 Index of Interface Items

Item	Page
AddEntry: PROCEDURE	3
ConvertProc: TYPE	2
Create: PROCEDURE	1
Destroy: PROCEDURE	2
Divider.ConvertProc:TYPE	2
DividerConvertProc: ConvertProc	2
DividerGenericProc: GenericProc	2
FindEntry: PROCEDURE	3
FindOrAddEntry: PROCEDURE	3
GenericProc: TYPE	2
Handle: TYPE	1
Object: TYPE	1

26

Event

26.1 Overview

ViewPoint provides a facility that permits clients to register procedures that are to be called when specified events occur. For example, a client may wish to be notified whenever a document is closed, or perhaps just the next time a document is closed. Clients need not know which module can cause the event.

26.2 Interface Items

26.2.1 Registering Dependencies

A client wishing to be notified of some future event calls either **AddDependency** or **AddDependencies**, specifying the **EventType** and an **AgentProcedure** to be called when the event occurs. Note: ViewPoint need not know in advance what **EventType** is implemented, nor which modules implement them.

```
AddDependency: PROCEDURE [
   agent: AgentProcedure,
   myData: LONG POINTER TO UNSPECIFIED,
   event: EventType,
   remove: FreeDataProcedure ← NIL]
   RETURNS [dependency: Dependency];

AddDependencies: PROCEDURE [
   agent: AgentProcedure,
   myData: LONG POINTER TO UNSPECIFIED,
   events: LONG DESCRIPTOR FOR ARRAY OF EventType,
   remove: FreeDataProcedure ← NIL]
   RETURNS [dependency: Dependency];

AgentProcedure: TYPE = PROCEDURE [
   event: EventType,
   eventData, myData: LONG POINTER TO UNSPECIFIED]
   RETURNS [remove, veto: BOOLEAN ← FALSE];
```

FreeDataProcedure: TYPE = PROCEDURE [mydata: LONG POINTER TO UNSPECIFIED];

Dependency: TYPE [2]; -- *Opaque* --

A dependency may be added to an event or an entire set of events by calling **AddDependency** or **AddDependencies**. Both of these procedures return a private type, **Dependency**, that uniquely identifies that set of dependencies. The value returned may be saved and subsequently used in a call to **RemoveDependency**, which removes the dependency or dependencies associated with the earlier **AddXXX** call. The **AgentProcedure** may also remove the dependency, as discussed below.

When the specified **event** occurs, **agent** is called with the **EventType**, the **eventData** for the event, and the client data passed as **myData**. If a client wishes to veto the event (for instance, to disallow a world-swap), its **AgentProcedure** should return **veto:** TRUE. This aborts the notification; that is, no other clients dependent on the event are notified. However, there is no guarantee of the order in which multiple clients are notified. If any client vetoes the event, the call to **Notify** returns TRUE. There is no way to prevent a client from vetoing; instead, implementors of events that should not be vetoed should raise an ERROR if **Notify** returns TRUE. To remove its dependency on an **event**, a client's **AgentProcedure** should return **remove:** TRUE. If the dependency is removed and a **FreeDataProcedure** was provided, it is called at this time to allow the client to free any private data.

EventType: TYPE = Atom.ATOM;

The **ATOM** (strings) used to identify different events must of course be distinct. The following examples are possibilities of how this could be managed. (1) By a central authority whose job it is to guarantee uniqueness of **EventTypes**. This could be the same person in charge of other such allocations, such as **NSFile** types. (2) By a hierarchical naming structure, managed by a distributed authority. (3) By a file that lists all known **EventTypes** within a given system; this file is managed by the Librarian to ensure against parallel allocation of new **EventTypes**. (In effect, this is the same as case 1, but the Librarian takes the place of the central authority.)

RemoveDependency: PROC [dependency: Dependency];

NoSuchDependency: ERROR;

If **RemoveDependency** is called with a **Dependency** that is invalid (possibly because the dependency has already been removed), it raises the error **NoSuchDependency**.

26.2.2 Notification

Notify: PROCEDURE [event: EventType, eventData: LONG POINTER TO UNSPECIFIED ← NIL]
  RETURNS [veto: BOOLEAN];

When the event occurs, the implementor calls **Notify**, giving it the **EventType** for the event and any implementation-specific data (**eventData**) required by the client. (Presumably it is uncommon for a single operation to wish to **Notify** more than one event; this is why **Notify** does not take an ARRAY argument.) The **Event** interface then invokes each **AgentProcedure** that is dependent on the **EventType**. Each **AgentProcedure** is given the

EventType causing the notification, the client data provided when the dependency was created, and the **eventData** given by the implementor in the call to **Notify**.

26.3 Usage/Examples

The **Event** database is monitored to disallow changes while a **Notify** is in progress. An **AgentProcedure** is allowed to call **Notify**; that is, one event may trigger another. However, an **AgentProcedure** must *not* call **AddDependency** or **RemoveDependency**, or deadlock will result. Because it is relatively common for an **AgentProcedure** to wish to remove its own dependency, the **AgentProcedure** can return **remove: TRUE** to cause the dependency to be removed. If the dependency was added via **AddDependencies**, then all of the dependencies created by that call are removed. The dependency is removed even though some later client of the same event might choose to veto the event. (If an earlier client has already vetoed, of course, then this **AgentProcedure** never gets called.) If an application requires that a dependency be removed only if the event is not vetoed, the implementor can notify a second event that informs clients whenever the first event is vetoed.

Three notes regarding the preceding paragraph: First, an **AgentProcedure** may get called twice even if it always returns **remove: TRUE** because two separate processes may be doing parallel calls to **Notify**. Once an **AgentProcedure** returns **remove: TRUE**, no subsequent calls to **Notify** invoke that dependency, but any parallel calls in progress complete normally. Second, because an **AgentProcedure** might be invoked at any time, it is a bad idea to call **Add/RemoveDependency** from within a private monitor, lest it lock trying to modify the **Event** database while a **Notify** is inside the **AgentProcedure** trying to grab the lock. However, the **Notify** call may very well be within the implementor's monitor, which means the **AgentProcedure's** use of the **eventData** is typically limited. Finally, if an **AgentProcedure** needs to call **Add/RemoveDependency**, it may get the desired effect by **FORK**ing the call so that it takes place shortly after the **Notify** already in progress.

26.3.1 Example 1

```
-- Module interested in an event
eventType: Event.EventType ← Atom.MakeAtom ["SampleEvent"L];

EventAction: Event.AgentProcedure = {
   -- Do appropriate thing for eventType -- };

Event.AddDependency [
   agent: EventAction,
   myData: NIL,
   event: eventType];

-- Module that signals the event
eventType: Event.EventType ← Atom.MakeAtom ["SampleEvent"L];
eventData: -- Relevant info, a record, a window handle, etc. --;
.
.
.
[] ← Event.Notify [event: eventType, eventData: eventData];
```

26.3.2 Example 2

```
-- Declare event and eventData --
desktopWindowAvailable: Event.EventType;
desktopWindowHandle: Window.Handle ← NIL;

-- Declare AgentProcedure --
StartUp: Event.AgentProcedure = {
   If eventData = NIL THEN RETURN [veto: TRUE];
   desktopWindowHandle ← eventData };

-- Register event-- this is mainline code --
[] ← Event.AddDependency [StartUp, NIL, desktopWindowAvailable];

-- In Desktop code, another module, notify occurrence of the event --
[] ← Event.Notify [desktopWindowAvailable, window];
 -- Window is desktop window --
```

26.4 Index of Interface Items

27 FileContainerShell

27.1 Overview

FileContainerShell provides a simple way to implement a container application that is backed by an NSFile. FileContainerShell takes an NSFile and column information (such as headings, widths, formatting) and creates a FileContainerSource, a StarWindowShell, and a ContainerWindow body. (See also the FileContainerSource, ContainerSource, StarWindowShell, and ContainerWindow interfaces). Most NSFile-backed container applications can use this interface, thereby greatly simplifying the writing of applications such as Folders and File Drawers.

27.2 Interface Items

27.2.1 Create a FileContainerShell

```
CreateX3: PROCEDURE [
   file: NSFile.Reference,
   columnHeaders: ContainerWindow.ColumnHeaders,
   columnContents: FileContainerSource.ColumnContents,
   regularMenuItems, topPusheeMenuItems: MenuData.ArrayHandle ← NIL,
   scope: NSFile.Scope ← [],
   position: ContainerSource.ItemIndex ← 0,
   options: FileContainerSource.Options ← [],
   access: ContainerWindowExtra3.Access ← ContainerWindowExtra3.fullAccess,
   considerShowingCoverSheet: BOOLEAN]
   RETURNS [shell: StarWindowShell.Handle];

CreateX: PROCEDURE [
   file: NSFile.Reference,
   columnHeaders: ContainerWindow.ColumnHeaders,
   columnContents: FileContainerSource.ColumnContents,
   regularMenuItems, topPusheeMenuItems: MenuData.ArrayHandle ← NIL,
   scope: NSFile.Scope ← [],
   position: ContainerSource.ItemIndex ← 0,
   options: FileContainerSource.Options ← [],
    access: ContainerWindowExtra3.Access ← ContainerWindowExtra3.fullAccess]
    RETURNS [shell: StarWindowShell.Handle];

Create: PROCEDURE [
    file: NSFile.Reference,
    columnHeaders: ContainerWindow.ColumnHeaders,
    columnContents: FileContainerSource.ColumnContents,
    regularMenuItems, topPusheeMenuItems: MenuData.ArrayHandle ← NIL,
    scope: NSFile.Scope ← [],
    position: ContainerSource.ItemIndex ← 0,
    options: FileContainerSource.Options ← []]
    RETURNS [shell: StarWindowShell.Handle];
```

Create, **CreateX**, and **CreateX3** create a **StarWindowShell** with a container window as the body window. **file** is the backing for the container; it must be an **NSFile** with children. **columnHeaders** and **columnContents** specify all the necessary information about the columns to be displayed for the open container. (See the **ContainerWindow** and **FileContainerSource** interfaces for the specifics of the headers and contents.) **scope** specifies ordering, filtering, and direction, if any. **position** indicates the item that should be displayed first. **access** specifies the **ContainerWindow** access. (See the **ContainerWindow** interface for details. **regularMenuItems** and **topPusheeMenuItems** are the menu items that the client would like to put in the header of the **StarWindowShell**. **Create** puts these items in the header along with its own menu items, such as Show Next and Show Previous. **considerShowingCoverSheet** specifies whether the resulting shell will be allowed to have a coversheet. TRUE means that the shell will have a coversheet if one is define; FALSE means that the coversheet will not be shown and the menu item "Show Coversheet" is not displayed. Fine point: The client is responsible for putting any **bottomPusheeCommands** in the window header. **CreateX** is defined in **FileContainerShellExtra.mesa**, **CreateX3** is defined in **FileContainerShellExtra3.mesa**

27.2.2 Operations on the Shell

```
GetContainerWindow: PROCEDURE [shell: StarWindowShell.Handle]
    RETURNS [window: Window.Handle];
```

Returns the container window that was created by the **Create** procedure. May raise ContainerWindow.Error[notAContainerWindow] if the shell does not have a container window in it.

```
GetContainerSource: PROCEDURE [shell: StarWindowShell.Handle]
    RETURNS [source:ContainerSource.Handle];
```

Returns the container source that was created by the **Create** procedure. May raise ContainerWindow.Error[notAContainerWindow] if the shell does not have a container window in it.

27.3 Usage/Examples

27.3.1 Example: Creating a FileContainerShell and Specifying Columns

The following example presents the procedure **CreateFileSWS**, which takes an **NSFile.Reference** and creates a file container shell with two columns: the name of the file and a version date. (See the **ContainerSource** interface for details on columns.) The name column uses the predefined **ContainerSource.NameColumn**; the version column is given in the example. The version column differs from the standard **ContainerSource.DateColumn** in that it displays the last modified date for directories instead of ---.

```
ContentSeq: TYPE = RECORD [
   SEQUENCE cols: CARDINAL OF FileContainerSource.ColumnContentsInfo];
HeaderSeq: TYPE = RECORD [
   SEQUENCE cols: CARDINAL OF ContainerWindow.ColumnHeaderInfo];
NumberOfColumns: CARDINAL = 2;
z: UNCOUNTED ZONE = ...;

CreateFileSWS: PROCEDURE [reference: NSFile.Reference]
   RETURNS [StarWindowShell.Handle] =
   BEGIN
   shell: StarWindowShell.Handle;
   headers: LONG POINTER TO HeaderSeq ← MakeColumnHeaders[];
   contents: LONG POINTER TO ContentSeq ← MakeColumnContents[];
   shell ← FileContainerShell.Create[
      file: reference,
      columnHeaders: DESCRIPTOR[headers],
      columnContents: DESCRIPTOR[contents]];
   z.FREE[@headers];
   z.FREE[@contents];
   RETURN[shell];
   END;

DateFormatProc: FileContainerSource.MultiAttributeFormatProc =
   BEGIN
   -- If non-directory, show createdOn date. For directory, show last date modified
        (the last time anything was changed in directory) --
   template: XString.ReaderBody ←
      XString.FromSTRING["<2>-<6>-<4> <8>:<9>:<10>"L];
   XTime.Append[
      displayString,
      IF attrRecord.isDirectory THEN attrRecord.modifiedOn ELSE attrRecord.createdOn,
      @template]};
   END;

MakeColumnContents: PROCEDURE
   RETURNS [columnContents: LONG POINTER TO ContentSeq] =
   BEGIN
   dateSelections: NSFile.Selections ← [interpreted: [
      isDirectory: TRUE, createdOn: TRUE, modifiedOn: TRUE]];
   columnContents ← z.NEW[ContentSeq[NumberOfColumns];
   columnContents[0] ← FileContainerSource.NameColumn[];
   columnContents[1] ← [multipleAttributes [attrs: dateSelections, formatProc:
DateFormatProc]];
   RETURN [columnContents];
   END;

MakeColumnHeaders: PROCEDURE
   RETURNS [columnHeaders: LONG POINTER TO HeaderSeq] =
   BEGIN
   columnHeaders ← z.NEW[HeaderSeq[NumberOfColumns]];
   columnHeaders[0] ← [
      width: 367,
      heading: XString.FromSTRING["NAME"] ];
   columnHeaders[1] ← [
      width: 135,
      heading: XString.FromSTRING["VERSION OF"] ];
   RETURN [columnHeaders];
   END;
```

27.4 Index of Interface Items

28

FileContainerSource

28.1 Overview

FileContainerSource supports the creation of **NSFile**-backed container sources (see **ContainerSource**). It also provides facilities for specifying the columns that will be displayed for each item in the source.

FileContainerSource implements all of the procedure types described in the **ContainerSource** interface, as well as all the procedures described below.

28.2 Interface Items

28.2.1 Creation

```
Options: TYPE = RECORD [
  readOnly: BOOLEAN ← FALSE];

Create: PROCEDURE [
  file: NSFile.Reference,
  columns: ColumnContents,
  scope: NSFile.Scope ← [],
  options: Options ← [] ]
  RETURNS [source: ContainerSource.Handle];

CreateX: PROCEDURE [
  file: NSFile.Reference,
  columns: ColumnContents,
  scope: NSFile.Scope ← [],
  options: Options ← [] ]
  RETURNS [source: ContainerSource.Handle, sourceX: ContainerSourceExtra.Procedures];
```

Creates a container source backed by **file**, which must be an **NSFile** with children. **columns** describes the information that should be displayed for each entry in the container. **columns** is copied by this procedure, so the client may release any storage associated with **columns** after calling **Create**. **scope** specifies the range of files that will be displayed. The caller of **Create** is responsible for the storage in the **scope** parameter; FileContainerSource will not copy it. It can be destroyed at the same time the source is destroyed. Typically the client

will save the pointer to **scope** storage in same place as **source** handle. **options** specifies global information about the container source. Display formatting is managed by the container window. (See the **ContainerWindow** and **FileContainerShell** interfaces.) **CreateX** is identical to **Create**, except that it returns the additional **ContainerSourceExtra.Procedures** needed for concurrency. (See the **ContainerSource** chapter for details on these procedures).
Fine point: CreateX is exported by FileContainerSourceExtra3.

28.2.2 Specifying Columns

When a file container source is created, columns may be specified. Each column represents information that will be displayed for each item. The container window requests the columns one at a time in the form of strings. In a file container source, each column must be based on some combination of **NSFile** attributes. For each column, the creator of file container source specifies which attributes are required to format a string for that column and supplies a procedure that will be called with the specified attributes. When the files in the source are enumerated, the procedure for a particular column is called with the values of the specified attributes for each file, which should be used to generate the string for that file.

```
ColumnContents: TYPE =
  LONG DESCRIPTOR FOR ARRAY OF ColumnContentsInfo;
```

ColumnContents describes a set of columns, where each column is some information that is displayed for each item in the container display. The columns are displayed in the order given by this array.

```
ColumnType:TYPE = {attribute, extendedAttribute, multipleAttributes};

ColumnContentsInfo: TYPE = RECORD [
  info: SELECT type: ColumnType FROM
    attribute = > [
      attr: NSFile.AttributeType,
      formatProc: AttributeFormatProc ← NIL],
    needsDataHandle: BOOLEAN ← FALSE],
      extendedAttribute = > [
      extendedAttr: NSFile.ExtendedAttributeType,
      formatProc: AttributeFormatProc ← NIL,
    extendedAttribute = > [
      extendedAttr: NSFile.ExtendedAttributeType,
      formatProc: AttributeFormatProc ← NIL],
     multipleAttributes = > [
      attrs: NSFile.Selections,
      formatProc: MultiAttributeFormatProc ← NIL],
    ENDCASE];
```

ColumnContentsInfo describes a single column of information that can be displayed for each item in a container display. Each column may be backed by one of three things: an **NSFile** interpreted attribute (the **attribute** variant), and **NSFile** extended attribute (the **extendedAttribute** variant), or some combination of several attributes (the **multipleAttributes** variant). The **attribute** and **extendedAttribute** variants both take a specification of what attribute is being described (**attr** and **extendedAttr**) and an **AttributeFormatProc** that is called to render the attribute as a string. If **needsDataHandle**

= TRUE, then a valid Containee.DataHandle is passed to the format procedure as the containeeData parameter, else the containeeData parameter is NIL. If the column needs a Containee.DataHandle in order to format it, then needsDataHandle should be TRUE. This addition is for performance: obtaining a Containee.DataHandle requires an extra access to the file, thus slowing up the enumeration. The multipleAttributes variant is for columns that may require more than one attribute. (The typical example is the SIZE column in folders, in which some items display the numberOfChildren attribute and others display the sizeInPages attribute, depending on the isDirectory attribute.) attrs specifies all the attributes required for this column. formatProc is the procedure that will be called to format the column.

See the common types of columns provided below in the section on commonly used columns.

```
AttributeFormatProc: TYPE = PROCEDURE [
   containeeImpl: Containee.Implementation,
   containeeData: Containee.DataHandle,
   attr: NSFile.Attribute,
   displayString: Xstring.Writer];
```

When the container display mechanism displays a column that represents an NSFile attribute, it calls the AttributeFormatProc specified for that column. attr contains the attribute to be formatted for display. displayString is used to return a formatted string that represents the desired attribute. containeeImpl may be used to make calls on the underlying implementation of the item being displayed.

```
MultiAttributeFormatProc: TYPE = PROCEDURE [
   containeeImpl: Containee.Implementation,
   containeeData: Containee.DataHandle,
   attrRecord: NSFile.Attributes, -- LONG POINTER TO NSFile.AttributesRecord
   displayString: Xstring.Writer];
```

When the container display mechanism displays a column that represents multiple NSFile attributes, it calls the MultiAttributeFormatProc specified for that column. attrRecord contains the attributes to be formatted for display. displayString is used to return a formatted string that represents the desired attribute. containeeImpl may be used to make calls on the underlying implementation of the item being displayed.

28.2.3 Operations on Sources

```
GetItemInfo: PROCEDURE [
   source: ContainerSource.Handle, itemIndex: ContainerSource.ItemIndex]
   RETURNS [file:NSFile.Reference, type: NSFile.Type];
```

Returns an NSFile.Reference and type for the specified item.

```
Info: PROCEDURE [source: ContainerSource.Handle]
   RETURNS [
   file: NSFile.Reference,
   columns: ColumnContents,
   scope: NSFile.Scope,
   options: Options];
```

```
InfoX: PROCEDURE[
   source: ContainerSource.Handle]
   RETURNS [sourceX: ContainerSourceExtra.Procedures];
```

The **Info** procedure returns information about a file container source; the information returned is the same information that was used to create the source (see the **Create** procedure). If the source was created using **CreateX**, **InfoX** returns the extra procedures defined in **ContainerSourceExtra.Procedures**. **Fine point: InfoX is defined in ContainerSourceExtra.**

```
IsIt: PROCEDURE [source: ContainerSource.Handle] RETURNS [BOOLEAN];
```

IsIt returns TRUE if **source** is a file container source.

```
ChangeScope: PROCEDURE [source: ContainerSource.Handle, newScope: NSFile.Scope];
```

Allows the scope (passed in to **Create**) to be changed. A call to **ChangeScope** is typically followed by a **source.ActOn[relist]**, then a **ContainerWindow.Update**.

```
RebuildItem: PROCEDURE [source: ContainerSource.Handle, item: ContainerSource.ItemIndex];
```

RebuildItem causes the **FileContainerSource** to rebuild **item**, for example after a client has changed an attribute that is displayed in a column of the source. Note that the client must call the appropriate ChangeProc in order to get the ContainerWindow to repaint properly. **Fine point: RebuildItem is exported by FileContainerSourceExtra2.**

```
SourceEnumProc: TYPE = PROCEDURE [source: ContainerSource.Handle]
   RETURNS [stop: BOOLEAN ← FALSE];

EnumerateSources: PROCEDURE [enumProc: SourceEnumProc];
```

EnumerateSources will enumerate all existing FileContainerSources and call **enumProc** with each source. The enumerate will stop early if the **enumProc** sets **stop** to TRUE. **Fine point: EnumerateSources and SourceEnumProc are defined in FileContainerSourceExtra3.**

```
TakeFilterProc: TYPE = PROCEDUER [fs: ContainerSource.Handle, aboutToTake: NSFile.Reference]
   RETURNS [ok: BOOLEAN];

SetTakeFilterProc: PROCEDURE : [fs: ContainerSource.Handle, p: TakeFilterProc];
```

Clients can use **SetTakeFilterProc** to set a **TakeFilterProc** that will be called just before each file is about to be moved or copied into the source. The **TakeFilterProc** returns a boolean; if ok is **TRUE**, the move or copy goes ahead, otherwise that file is not copied/moved into the source and the enumeration continues on to the next file. No message is posted if the **TakeFilterProc** vetos the copy, so feedback is up to the client. This procedure is provided to allow clients some control over what files are copied into the source; clients may use this to make sure their source only gets files of certain types. This proc has no effect on copies onto the closed container; clients must set up a separate filter mechanism for that case. For BWS 4.3, these two procedures are in FileContainerSourceExtra4.

28.2.4 Commonly Used Columns

These predefined procedures can be used in building a **ColumnContents** array.

```
IconColumn: PROCEDURE
  RETURNS [attribute ColumnContentsInfo];
```

IconColumn represents a column with a small icon picture in it. The small picture is obtained from the **containeeImpl.smallPicture** that is passed in.

```
NameColumn: PROCEDURE
  RETURNS [attribute ColumnContentsInfo];
```

NameColumn represents a column with the file's name in it.

```
SizeColumn: PROCEDURE
  RETURNS [multipleAttributes ColumnContentsInfo];
```

SizeColumn represents a column with the file's size in it, as follows: If the file has the **isDirectory** attribute, the **numberOfChildren** attribute is displayed with the label "Objects"; if the file does not have the **isDirectory** attribute, the **sizeInPages** attribute is displayed with the label "Disk Pages".

```
DateColumn: PROCEDURE
  RETURNS [multipleAttributes ColumnContentsInfo];
```

DateColumn represents a column with the file's creation date in it, as follows: If the file has the **isDirectory** attribute, dashes (---) are displayed; if the file does not have the **isDirectory** attribute, the **createDate** attribute is displayed.

```
VersionColumn: PROCEDURE
  RETURNS [attribute ColumnContentsInfo];
```

VersionColumn represents a column with the file's version in it. **VersionColumn** is defined in FileContainerSourceExtra.mesa.

```
NameAndVersionColumn: PROCEDURE
  RETURNS [multipleAttributes ColumnContentsInfo];
```

NameAndVersionColumn represents a column with the file's name and version appended together with an exclamation point in between, e.g. Foo!3. **NameAndVersionColumn** is defined in FileContainerSourceExtra.mesa.

28.3 Usage/Examples

28.3.1 Example: Specifying Columns using FileContainerSource

The following example presents the procedure **MakeFolderLikeShell**, which takes an **NSFile.Reference** (**Containee.DataHandle**) and creates a file container shell with the number of columns dependent on some internal procedures. (See the **ContainerSource** interface for details on columns.) The columns use the predefined columns such as ContainerSource.NameColumn.

```
Columns: TYPE = {icon, name, version, nameAndVersion, size, createDate};
HeaderSeq: TYPE = RECORD [SEQUENCE cols: CARDINAL OF ContainerWindow.ColumnHeaderInfo];
ContentSeq: TYPE = RECORD [
   SEQUENCE cols: CARDINAL OF FileContainerSource.ColumnContentsInfo];
ColumnArray:TYPE = ARRAY {icon, name, version, size, date} OF CARDINAL;
columnWidths: LONG POINTER TO ColumnArray ← z.NEW[ColumnArray ← NULL];

ClientsGenericProc: Containee.GenericProc =
   <<[atom: Atom.ATOM,
   data: Containee.DataHandle,
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [LONG UNSPECIFIED]>>
   BEGIN
      SELECT atom FROM
         open => RETURN [
               MakeFolderLikeShell [
                     data: data,
                     changeProc: changeProc,
                     changeProcData: changeProcData] ];
      .
      .
      .
         ENDCASE => RETURN [ oldFolder.genericProc [atom, data] ];
      END;

FreeColumnContents: PUBLIC PROCEDURE [columnContents: LONG POINTER TO ContentSeq] =
   BEGIN
   z.FREE[@columnContents];
   END;

FreeColumnHeaders: PUBLIC PROCEDURE [columnHeaders: LONG POINTER TO HeaderSeq] =
BEGIN
z.FREE[@columnHeaders];
END;

MakeFolderLikeShell: PROCEDURE [
   data: Containee.DataHandle,
   changeProc: Containee.ChangeProc ← NIL,
   changeProcData: LONG POINTER ← NIL]
   RETURNS [shell: StarWindowShell.Handle] = {
   file: NSFile.Reference;
   columnHeaders: LONG POINTER TO HeaderSeq ← MakeColumnHeaders[];
   columnContents: LONG POINTER TO ContentSeq ← MakeColumnContents[];
   .
   .
   .
   mydata: Data ← z.NEW [DataObject ← [
      cd: data,
    changeProc: changeProc,
    changeProcData: changeProcData]];
  isLocal: BOOLEAN;
  BEGIN ENABLE
    UNWIND => {
      z.FREE[@mydata];
      FreeColumnHeaders [columnHeaders];
      FreeColumnContents [columnContents];
      };
  .
  .
  .
  shell ← FileContainerShell.Create [
    file: file,
    columnHeaders: DESCRIPTOR[columnHeaders],
    columnContents: DESCRIPTOR[columnContents],
    regularMenuItems: IF ~isLocal THEN remoteRegularMenuItems ELSE NIL];

  IF shell = NIL THEN RETURN [shell];

  StarWindowShell.SetIsCloseLegalProc [shell, Closing];
  Context.Create[context, mydata, DestroyContext, shell];
  FreeColumnHeaders [columnHeaders];
  FreeColumnContents [columnContents];
  StarWindowShell.SetPreferredDims [ shell, [700, 0] ];

  RETURN [shell];
  END; -- ENABLE
  }

MakeColumnContents: PUBLIC PROCEDURE RETURNS [columnContents: LONG POINTER TO
ContentSeq] =
  BEGIN
  i: INTEGER ← -1;
  columnContents ← z.NEW[ContentSeq[CountColumns[]]];
  IF ShowIcon[] THEN
    columnContents[i ← i + 1] ← FileContainerSource.IconColumn[];
  -- Procedures called below are not neccessary to the example.
  columnContents[i ← i + 1] ←
    IF ShowNameAndVersion[]
      THEN FileContainerSourceExtra.NameAndVersionColumn[]
    ELSE FileContainerSource.NameColumn[];
  IF ShowVersion[] THEN
    columnContents[i ← i + 1] ← FileContainerSourceExtra.VersionColumn[];
  IF ShowSize[] THEN
    columnContents[i ← i + 1] ← FileContainerSource.SizeColumn[];
  IF ShowCreateDate[] THEN
    columnContents[i ← i + 1] ← FileContainerSource.DateColumn[];
  RETURN [columnContents];
  END;
```

28.4 Index of Interface Items

29

FormWindow

29.1 Overview

The **FormWindow** interface provides clients the ability to create and manipulate form items in a window.

There are several types of items, each of which serves a different purpose and behaves differently for the user. All items except **tagonly** and **command** have a current value that can be obtained and set by the client and user. The user obtains the current value of an item by simply looking at it and sets the current value of an item by pointing at it appropriately with the mouse. The client obtains and sets the value of items by calling appropriate **FormWindow** procedures.

A **boolean** item is an item with two states (on and off, or TRUE and FALSE). A boolean item's value is of type **BOOLEAN**.

A **choice** item has an enumerated list of choices, only one of which can be selected at any point in time. A choice item's value is of type **FormWindow.ChoiceIndex**.

A **multiplechoice** item is a **choice** item that can have an initial value of more than one choice selected, but any succeeding values can have only one choice selected. A **multiplechoice** item's value is of type **LONG DESCRIPTOR FOR ARRAY OF CARDINAL**.

A **text** item is a user-editable text string. It contains nonattributed text only. A text item's value is of type **XString.ReaderBody**.

A **decimal** item is a text item that has a value of type **XLReal.Number**.

An **integer** item is a text item that has a value of type **LONG INTEGER**.

A **command** item allows a user to invoke a command. When the user clicks over a command item, a client procedure is called.

A **tagonly** item is an uneditable, nonselectable text string.

A **window** item is a window that is a child of the **FormWindow**. It can contain whatever the client desires. A window item's value is a **Window.Handle**. A client must provide its own **TIP.NotifyProc** and window display proc for the window item.

29.1.1 Creating a FormWindow

A client creates a **FormWindow** by calling **FormWindow.Create**. **Create** does not actually create a window, but rather it takes an already existing window and turns it into a **FormWindow**. **Windows** are usually created by calling **StarWindowShell.CreateBody**.

The client supplies a **MakeItemsProc** and optionally a **LayoutProc** to **FormWindow.Create**. **Create** calls these two client procedures, first the **MakeItemsProc**, then the **LayoutProc**. In the **MakeItemsProc**, the client creates the individual items in the form by calling FormWindow procedures that make items (see §29.1.2 and §29.2.2). In the **LayoutProc**, the client specifies where each created item should be positioned in the window by calling FormWindow procedures that specify layout (see the sections labeled Layout in this chapter).

29.1.2 Making Form Items

There is a procedure for making each type of item: **MakeBooleanItem**, **MakeChoiceItem**, **MakeCommandItem**, **MakeDecimalItem**, **MakeIntegerItem**, **MakeMultipleChoiceItem**, **MakeTagOnlyItem**, **MakeTextItem**, **MakeWindowItem**. Each item must have a unique "key", a **FormWindow.ItemKey**. This is a **CARDINAL** supplied by the client to each **MakeXXXItem** call. This key is then used in any future calls to manipulate that item, such as to get the value of the item. The key must be unique within the **FormWindow**.

All items have some common characteristics and some type-unique characteristics. The common ones are described here. Every item can have a tag that will appear to the left of the item and a suffix that will appear to the right of the item. An item can have a box drawn around it or not. The default is to draw the box. Items can be read-only, that is the user cannot change the value of the item. Items can be visible or invisible, and invisible items can either take up white space in the window or not. See §29.2.2 for more details.

29.1.3 Getting and Setting Values

Every item that has a value that the user can change (all except tagonly and command items) also has procedures for the client to get and set the value. These are:

GetBooleanItemValue	**DoneLookingAtTextItemValue**
GetChoiceItemValue	**SetBooleanItemValue**
GetDecimalItemValue	**SetChoiceItemValue**
GetIntegerItemValue	**SetDecimalItemValue**
GetMultipleChoiceItemValue	**SetIntegerItemValue**
GetTextItemValue	**SetMultipleChoiceItemValue**
GetWindowItemValue	**SetTextItemValue**
LoookAtTextItemValue	

Note: All allocation of storage for values of items is handled by **FormWindow**. The client need not keep copies of item values while the **FormWindow** exists. Obtaining the current value of an item is a simple call to one of the **GetXXXItemValue** procedures. This makes it easy to ensure that the internal value of an item is always in sync with the display. (See §29.2.3 for more details.) Fine Point: This storage allocation scheme is opposite to the one used by XDE's FormSW, where the client owns the storage for items.

29.1.4 "Changed" BOOLEAN

Every item that has a value that the user can change (all except **tagonly**, **command**, and **window items**) has a "changed" boolean associated with it. All items are created with this boolean set to FALSE. **FormWindow** automatically sets this boolean to TRUE whenever the user changes the item. This allows the client to determine which items have changed when, for example, the user selects "Done" or "Apply" on a property sheet. The client is responsible for resetting the changed boolean to false by calling **ResetChanged** or **ResetAllChanged** after examining the changed boolean with **HasBeenChanged** or **HasAnyBeenChanged**. See §29.2.1 for more detail.

Boolean and choice items can have a client-supplied procedure that will be called whenever the item's value changes (see **BooleanChangeProc** and **ChoiceChangeProc** in §29.2.1 and 29.2.2. The client may also supply a **GlobalChangeProc** that will be called whenever any item changes (see §29.2.1).

29.1.5 Visibility

Each item is either displayed in the form window or not. If an item is displayed in the form window, it is visible. If an item is not currently displayed, it is either invisible or **invisibleGhost**. If it is invisible, it does not take up any space on the screen, that is any items below it move up to take its screen space. If an item is **invisibleGhost**, the space that it would occupy were it visible is white on the screen. An item's visibility can be changed at any time by calling **SetVisibility** (see §29.2.5.)

29.1.6 Layout

The exact layout of items in a form window is done by calling various layout procedures after creating the items to be laid out. If an item is not explicitly laid out, it will not appear in the form window at all. A **DefaultLayout** procedure is provided that places each created item on a separate line.

A form window consists of horizontal lines with zero or more items on each line. Each line may be a different height. Any desired vertical spacing may be accomplished by using appropriate heights for lines. Any desired horizontal spacing may be accomplished by using appropriate margins between items. Items may be lined up horizontally by using **TabStops**. Lines are created by calling **AppendLine** or **InsertLine**. Items are placed on a line by calling **AppendItem** or **InsertItem**. (See §29.2.6 for more detail.)

29.1.7 Neutral Properties

Any item (except command items) can take on a *neutral* state, which indicates that the item has no value at all. This property makes it possible for a client to indicate to the user that the item's value is no longer valid in the current context of the form window, or that the value for this item is obtained from somewhere other then this form window. Items in the neutral state appear without any indications of value and have their tag, suffix, and value box fields painted over with gray diagonal stripes. Since the item actually has no value, client calls to **GetXXXItemValue** for an item in the neutral state result in an error.

Items can be placed into (and removed from) a neutral state by calls to **SetItemNeutralness**. A neutral item automatically returns to a normal state when the user selects a value for that item, or when the item receives the text input focus (in the case of text, integer, and decimal items). (See §29.2.8 for more detail.)

29.2 Interface Items

29.2.1 Creating a FormWindow, etc.

```
Create: PROCEDURE[
   window: Window.Handle,
   makeItemsProc: MakeItemsProc,
   layoutProc: LayoutProc ← NIL,
   windowChangeProc: GlobalChangeProc ← NIL,
   minDimsChangeProc: MinDimsChangeProc ← NIL,
   zone: UNCOUNTED ZONE ← NIL,
   clientData: LONG POINTER ← NIL ];
```

Create takes an ordinary window and makes it a form window.

window is a window created by the client. Windows are usually created by calling **StarWindowShell.CreateBody**.

makeItems is a client-supplied procedure that is called to make the form items in the window. **makeItems** should call various **FormWindow.MakeXXXItem** procedures (see §29.2.2). Fine Point: **makeItems** is not called after **Create** returns, so **makeItems** can be a nested procedure.

layoutProc is a client-supplied procedure that is called to specify the desired position of the items in the window. **layoutProc** is called after **makeItems** has been called. **layoutProc** should call various layout procedures (see §29.2.6), such as **AppendLine** and **AppendItem**. If the default is taken, the **DefaultLayout** of one item per line will be used.

windowChangeProc is the global change proc for the entire window. Any time any item in the window changes, this procedure is called.

zone is the zone from which storage for the items will be allocated. **FormWindow** uses a private zone if none is supplied.

clientData is passed to **makeItems**, **layoutProc**, and **windowChangeProc** when called.

May raise **Error[alreadyAFormWindow]**.

```
DefaultLayout: LayoutProc;
```

The default for the **Create layoutProc** parameter. Specifies a layout of one item per line.

```
Destroy: PROCEDURE [window: Window.Handle];
```

Destroy destroys all **FormWindow** data associated with **window**, turning it back into an ordinary window. All form items are destroyed, but the window itself is not destroyed. May raise **Error[notAFormWindow]**.

```
GetClientData: PROCEDURE [window: Window.Handle]
  RETURNS [clientData: LONG POINTER];
```

GetClientData returns the **clientData** that was passed to **Create**. May raise **Error[notAFormWindow]**.

```
GlobalChangeProc: TYPE = PROCEDURE [
  window: Window.Handle,
  item: ItemKey,
  calledBecauseOf: ChangeReason,
  clientData: LONG POINTER];
```

The client may supply a **GlobalChangeProc** to **Create**. Any time the value of any item in the window is changed, the **GlobalChangeProc** is called with the key of the item that was changed. If more than one item was changed at one time (such as by a client call to **FormWindow.Restore**), **nullItemKey** will be passed in and the client must examine the "changed" boolean of all items to see what was changed (see §29.2.4). **calledBecauseOf** indicates what kind of action caused the **GlobalChangeProc** to be called. **clientData** is the **LONG POINTER** that was passed to **Create**.

```
GetGlobalChangeProc: PROCEDURE [window: Window.Handle]
  RETURNS [proc: GlobalChangeProc];
```

GetGlobalChangeProc returns the **GlobalChangeProc** that was passed to **Create**. May raise **Error[notAFormWindow]**.

```
SetGlobalChangeProc: PROCEDURE [window: Window.Handle,
  proc: GlobalChangeProc] RETURNS [old: GlobalChangeProc];
```

SetGlobalChangeProc changes the **GlobalChangeProc** that was passed to **Create**. May raise **Error[notAFormWindow]**.

```
MinDimsChangeProc: TYPE = PROCEDURE [window: Window.Handle,
  old, new: Window.Dims];
```

Whenever the minimum dimensions of the **FormWindow** change, the client supplied **MinDimsChangeProc** is called. This is useful for form windows that are nested as window items inside another outer form window. Whenever the dimensions of the nested form window change (due to items being made visible or invisible or a text item growing or shrinking or new items being added or...), the client that created the window item and the nested form window can be called so that it can make the window item bigger or smaller for the nested form window to be completely visible. See also **NeededDims**.

```
SetMinDimsChangeProc: PROCEDURE [window: Window.Handle,
  proc: MinDimsChangeProc] RETURNS [old: MinDimsChangeProc];
```

SetMinDimsChangeProc changes the **MinDimsChangeProc** that was passed to **Create**. May raise **Error [notAFormWindow]**. **SetMinDimsChangeProc** is defined in FormWindowExtra2.mesa.

GetZone: PROCEDURE [window: Window.Handle]
  RETURNS [zone: UNCOUNTED ZONE];

GetZone returns the zone associated with the **FormWindow**. May raise **Error[notAFormWindow]**.

IsIt: PROCEDURE [window: Window.Handle] RETURNS [yes: BOOLEAN];

IsIt is used to determine if a window is a form window. If **window** was made into a form window by calling **FormWindow.Create**, then **IsIt** returns **TRUE**, else **FALSE**.

LayoutProc:TYPE = PROCEDURE [window: Window.Handle, clientData: LONG POINTER];

The client supplies a **LayoutProc** to **Create** to specify the location of items created by the **MakeItemsProc**. See §29.2.6 for details of layout.

MakeItemsProc: TYPE = PROCEDURE [
  window: Window.Handle,
  clientData: LONG POINTER];

The client supplies a **MakeItemsProc** to **Create** to make the form items in the window. **Create** will call the client's **MakeItemsProc**, and it should call various **MakeXXXItem** procedures (see §29.2.2) to make the items. **window** should be passed to the various **MakeXXXItem**. **clientData** is the same as that passed to **Create**. Fine point for clients of **PropertySheet: clientData** can be passed to **PropertySheet.Create** and will be passed on to **FormWindow.Create** and the **MakeItemsProc**.

NeededDims: PROCEDURE [window: Window.Handle]
  RETURNS [Window.Dims];

NeededDims returns the minimum dimensions required for a window to hold all the currently visible items in the form.

NumberOfItems: PROCEDURE [window: Window.Handle] RETURNS [CARDINAL];

NumberOfItems returns the current number of form items in **window**. This count will include visible and invisible items. This is useful for clients that create additional items dynamically after the form has been created. May raise **Error[notAFormWindow]**.

Repaint: PROCEDURE [window: Window.Handle];

Repaint causes a **Window.Validate** on **window**. This is used in conjunction with the **SetXXXItemValue**, **SetVisibility**, **AppendItem**, and **InsertItem** procedures. All these procedures take a **repaint: BOOLEAN** parameter. To minimize screen flashing while changing several items at the same time, the client may call these procedures with repaint **= FALSE**, then call **FormWindow.Repaint**. The form window will not be repainted until Repaint is called. **Warning:** After calling any procedure with **repaint = FALSE**, **FormWindow.Repaint**

must be called. Otherwise, the screen will be inconsistent with the internal values. May raise **Error[notAFormWindow]**.

29.2.2 Making Form Items, etc.

Create procedures are provided for each type of item. These **MakeXXXItem** routines are used to originally create items in a form window as well as to add items to an existing window.

A number of parameters to each **MakeXXXItem** procedure are identical and are described here, rather than with each procedure. If all of the defaults are taken for an item, it will be boxed, with no tags and not read-only. All of these may raise **Error[notAFormWindow]**;

window is the form window the item is contained in. It should be the same as the window passed to the client's **MakeItemsProc**.

myKey is a client-defined key (**ItemKey**) for the item. The item key uniquely identifies the item and should be used to make calls on other **FormWindow** procedures, such as **GetXXXItemValue**. **Caution:** The key must be *unique* within this form window.

tag is the text to be displayed before (to the left of) the item on the same line. (To put a tag on a separate line, use **MakeTagOnlyItem**.)

suffix is the text to be displayed after (to the right of) the item on the same line.

visibility indicates whether the item should be displayed on the screen.

boxed indicates whether the item should have a box drawn around it or not.

readOnly = TRUE indicates that the item can not be edited by the user. The item can still be changed by calling a **SetXXXItemValue** procedure.

ItemKey: TYPE = CARDINAL;

ItemKey uniquely identifies an item. An **ItemKey** is supplied by the client whenever an item is made (**MakeXXXItem**) and should be used thereafter to identify the item to **FormWindow**, such as then calling **GetXXXItemValue** or **SetVisibility**.

ItemType: TYPE = MACHINE DEPENDENT **{choice(0), multiplechoice, decimal, integer, boolean, text, command, tagonly, window, last(15)};**

There are several types of items, each of which serves a different purpose and behaves differently for the user. All items except tagonly and command have a current value that can be obtained (**GetXXXItemValue**) and set (**SetXXXItemValue**).

A **choice** item has an enumerated list of choices, only one of which can be selected at any point in time. A choice item's value is of type FormWindow.**ChoiceIndex**.

A **multiplechoice** item is a choice item that can have an initial value of more than one choice selected, but any succeeding values can have only one choice selected. A multiple choice item's value is of type LONG DESCRIPTOR FOR ARRAY OF CARDINAL.

A **text** item is a user-editable text string, and contains only nonattributed text. A text item's value is of type XString.**ReaderBody**.

A **decimal** item is a text item that has a value of type XLReal.**Number**.

An **integer** item is a text item that has a value of type LONG INTEGER.

A **boolean** item is an item with two states (on and off, or TRUE and FALSE). A boolean item's value is of type **BOOLEAN**.

A **command** item allows a user to invoke a command. When the user clicks over a command item, a client procedure is called.

A **tagonly** item is an uneditable, nonselectable text string.

A **window** item is a window that is a child of the **FormWindow** and can contain whatever the client desires. A window item's value is a **Window.Handle**. A client must provide its own **TIP.NotifyProc** and window display procedure for the window item.

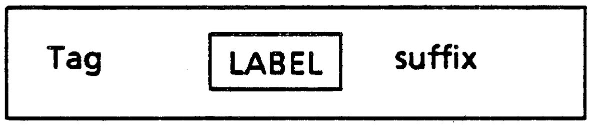
```
nullItemKey: ItemKey;
```

nullItemKey is used to indicate no item.

29.2.2.1 Boolean Items

```
MakeBooleanItem: PROCEDURE [
   window: Window.Handle,
   myKey: ItemKey,
   tag: XString.Reader ←NIL,
   suffix: XString.Reader ←NIL,
   visibility: Visibility ←visible,
   boxed: BOOLEAN ←TRUE,
   readOnly: BOOLEAN ←FALSE,
   changeProc: BooleanChangeProc ←NIL,
   label: BooleanItemLabel,
   initBoolean: BOOLEAN ←TRUE];
```

MakeBooleanItem creates a boolean item. A boolean item value is of type **BOOLEAN**. When the value is **TRUE**, the item is highlighted. When **FALSE**, it is not highlighted. When the user clicks over the label part of a boolean item, the value toggles.



Unhighlighted boolean item, value = FALSE

changeProc is a client-supplied procedure that will be called whenever the value of the item changes.

label is the string or bitmap that the user points at to toggle the item's value. If **label** is a string, the string is copied. If **label** is a bitmap, the bits are *not* copied, so the client must ensure that the bitmap pointer is valid for the lifetime of the form window.

initBoolean is the initial value of the item.

May raise **Error[notAFormWindow, duplicateItemKey]**.

```
BooleanItemLabel: TYPE = RECORD [
   var: SELECT type: BooleanItemLabelType FROM
      string = > [ string: XString.ReaderBody],
    bitmap => [bitmap: Bitmap]
    ENDCASE];
```

```
BooleanItemLabelType: TYPE = {string, bitmap};
```

A **BooleanItemLabel** is passed to **MakeBooleanItem**. It is the part of the item that the user points at and is or is not highlighted, depending on the value of the item. A **label** may be either a string or a bitmap. (See §29.2.10 on Miscellaneous **TYPEs** for the definition of **Bitmap**). If **label** is a string, the string is copied. If **label** is a bitmap, the bits are *not* copied, so the client must ensure that the bitmap pointer is valid for the lifetime of the form window.

```
BooleanChangeProc: TYPE = PROCEDURE [
  window: Window.Handle,
  item: ItemKey,
  calledBecauseOf: ChangeReason,
  newValue: BOOLEAN];
```

The client may provide a **BooleanChangeProc** to **MakeBooleanItem**. Whenever the item's value changes (**TRUE** to **FALSE** or **FALSE** to **TRUE**), this procedure is called. **window** is the form window that the item is in. **item** is the key of the boolean item to which this **BooleanChangeProc** is attached. **calledBecauseOf** indicates what kind of action caused the change proc to be called. **newValue** is the vew value of the item. The item will already have the new value when this procedure is called.

Caution: If a **BooleanChangeProc** does a **SetXXXItemValue**, the client should take extreme care to prevent infinite recursion. (See §29.3.1.)

29.2.2.2 Choice Items

```
MakeChoiceItem: PROCEDURE [
  window: Window.Handle,
  myKey: ItemKey,
  tag: XString.Reader ← NIL,
  suffix: XString.Reader ← NIL,
  visibility: Visibility ← visible,
  boxed: BOOLEAN ← TRUE,
  readOnly: BOOLEAN ← FALSE,
  values: ChoiceItems,
  initChoice: ChoiceIndex,
  fullyDisplayed: BOOLEAN ← TRUE,
  verticallyDisplayed: BOOLEAN ← FALSE,
  hintsProc: ChoiceHintsProc ← NIL,
  changeProc: ChoiceChangeProc ← NIL,
  outlineOrHighlight: OutlineOrHighlight ← highlight];
```

MakeChoiceItem creates a choice item. A choice item is an enumerated list of choices, only one of which can be selected at any time . The choices can be displayed to the user as either strings or bitmaps, or some of each. The current choice is highlighted. When the user clicks on a choice, it becomes the current choice and is highlighted. Each choice has a client-defined **ChoiceIndex** associated with it that uniquely identifies that choice. The value of a choice item is of type **ChoiceIndex**.

values is the list of all the possible choices. An indication of where to wrap the display around to the next line can be made by specifying a **wrapIndicator** variant in the appropriate place in the **values** array. If a choice is a string, the string is copied. If a choice is a bitmap, the bits are *not* copied, so the client must ensure that the bitmap pointer is valid for the lifetime of the form window.

initChoice is the value of the initial choice.

fullyDisplayed indicates whether all the choices should be displayed or not. If **fullyDisplayed** = TRUE, all the choices are displayed. If **fullyDisplayed** = FALSE, only the current choice is displayed, with the rest of the choices being accessed via a popup menu. **Caution**: bitmaps cannot appear in popup menus, so **fullyDisplayed** = FALSE should not be used if the choices are bitmaps.

verticallyDisplayed indicates whether the choices should be displayed vertically or horizontally. If **fullyDisplayed** = FALSE, the value of **verticallyDisplayed** is ignored. Any **wrapIndicators** are skipped over when choices are displayed vertically.

If **hintsProc** is supplied, it is called to make a popup hint menu. If the default is taken, the form window will make a hint menu with all choices. **Note:** Since menus can only contain strings (not bitmaps), a bitmap choice will appear in the hints menu as a number indicating the choice's position. **Note:** This is *not* the same as the **ChoiceIndex** for that choice.

If **changeProc** is supplied, it is called whenever the choice changes.

May raise **Error[notAFormWindow,duplicateItemKey, invalidChoiceNumber]**.

```
OutlineOrHighlight: TYPE = {outline, highlight};
```

Normally the selected choice for a choice item is indicated by highlighting the choice. The **outlineOrHighlight** parameter allows the selected choice to be indicated by outlining the choice with a black box. This is intended to support the Shading choice item on, for example, the triangle and ellipse property sheets in the ViewPoint editor.

```
ChoiceItems: TYPE = LONG DESCRIPTOR FOR ARRAY ChoiceIndex OF ChoiceItem;
```

ChoiceItems is the list of possible choice for a choice item. A **ChoiceItems** ARRAY is passed to **MakeChoiceItem**. The choices are displayed in the order they appear in the **ChoiceItems** ARRAY.

```
ChoiceItem: TYPE = RECORD [
  var: SELECT type: ChoiceItemType FROM
  string = > [
    choiceNumber: ChoiceIndex,
    string: XString.ReaderBody],
  bitmap = >[
    choiceNumber: ChoiceIndex,
    bitmap: Bitmap],
  wrapIndicator = > NULL];
```

```
ChoiceItemType: TYPE = {string, bitmap, wrapIndicator};
```

```
ChoiceIndex: TYPE = CARDINAL [ 0..37777B ];
```

A choice item consists of an array of choices (**ChoiceItems**). Each choice (**ChoiceItem**) consists of a unique number that identifies the choice (**ChoiceIndex**) and either a string or a bitmap to display to the user. In addition, the **ChoiceItems** array can contain a **wrapIndicator** wherever the client desires the choices be wrapped around to begin another line of choices. A **wrapIndicator ChoiceItem** is not a real choice and serves only as additional layout information for the **FormWindow**. If **ChoiceItem** is a string, the string is copied. If **ChoiceItem** is a bitmap, the bits are *not* copied, so the client must ensure that the bitmap pointer is valid for the lifetime of the **FormWindow**.

The client must construct a **ChoiceItems** array before calling **MakeChoiceItem**. This can be simplified if all the choices are strings by using the **FormWindowMessageParse** interface. This allows all the choices for a choice item to be stored as a single **XMessage** with embedded syntax indicating individual choice strings and choice numbers. (See **FormWindowMessageParse** for more detail.)

```
ChoiceChangeProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   calledBecauseOf: ChangeReason,
   oldValue, newValue: ChoiceIndex];
```

The client may provide a **ChoiceChangeProc** to **MakeChoiceItem**. Whenever the choice changes, this procedure is called. **window** is the form window that the item is in. **item** is the key of the choice item to which this **ChoiceChangeProc** is attached. **calledBecauseOf** indicates what kind of action caused the change proc to be called. **oldValue** and **newValue** correspond to the choice numbers assigned to the choices in **MakeChoiceItem**. The item will have the new value when this procedure is called.

Caution: If a **ChoiceChangeProc** does a **SetXXXItemValue**, the client should take extreme care to prevent infinite recursion. See §29.3.1, Calling **ChangeProcs**.

```
ChoiceHintsProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [
   hints: LONG DESCRIPTOR FOR ARRAY OF ChoiceIndex,
   freeHints: FreeChoiceHintsProc];

FreeChoiceHintsProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   hints: LONG DESCRIPTOR FOR ARRAY OF ChoiceIndex];
```

The client may provide a **ChoiceHintsProc** to **MakeChoiceItem**. Whenever the user points at the mouse menu for a choice item, this procedure is called and the hints returned are used to construct a popup menu that is displayed. If the user selects one of the choices from the popup menu, that choice becomes the current choice.

window is the form window that the item is in.

item is the key of the choice item to which this **ChoiceHintsProc** is attached.

hints is an array of choice numbers for the choices that the client wants to appear in the menu. This allows a client to show a subset of all the choices to the user for situations in which not all the choices make sense. **hints** must be allocated by the client.

freeHints is a procedure that will be called after the hint menu has been taken down to allow the client to free any storage that was allocated when creating the hints array.

```
MakeMultipleChoiceItem: PROCEDURE [
   window: Window.Handle,
   myKey: ItemKey,
   tag: XString.Reader ← NIL,
   suffix: XString.Reader ← NIL,
   visibility: Visibility ← visible,
   boxed: BOOLEAN ← TRUE,
   readOnly: BOOLEAN ← FALSE,
   values: ChoiceItems,
   initChoice: LONG DESCRIPTOR FOR ARRAY OF ChoiceIndex,
   verticallyDisplayed: BOOLEAN ← FALSE,
   hintsProc: ChoiceHintsProc ← NIL,
   changeProc: MultipleChoiceChangeProc ← NIL];
```

May raise **Error[notAFormWindow, duplicateItemKey]**.

```
MultipleChoiceChangeProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   calledBecauseOf: ChangeReason,
   oldValue: LONG DESCRIPTOR FOR ARRAY OF ChoiceIndex,
   newValue: LONG DESCRIPTOR FOR ARRAY OF ChoiceIndex];
```

A multiple choice item is identical to a choice item, except that it may have more than one initial value. See **MakeChoiceItem** above for details of choice items. A multiple choice item is useful for showing the properties of a heterogenous selection, such as the font property of a text selection that has more than one font.

29.2.2.3 Command Items

```
MakeCommandItem: PROCEDURE [
   window: Window.Handle,
   myKey: ItemKey,
   tag: XString.Reader ← NIL,
   suffix: XString.Reader ← NIL,
   visibility: Visibility ← visible,
   boxed: BOOLEAN ← TRUE,
   readOnly: BOOLEAN ← FALSE,
   commandProc: CommandProc,
   commandName: XString.Reader,
   clientData: LONG POINTER ← NIL];
```

Creates a command item. A command item allows a user to invoke a command. When the user clicks over the **commandName**, **commandProc** is called. If **boxed** is TRUE, the **commandName** appears with a rounded corner box drawn around it (rather than a squarecornered box, to distinguish a command item from a boolean item). May raise Error[notAFormWindow, duplicateItemKey].

```
NewMakeCommandItem: PROCEDURE [
  window: Window.Handle,
  myKey: ItemKey,
  tag: XString.Reader ←NIL,
  suffix: XString.Reader ←NIL,
  visibility: Visibility ←visible,
  boxed: BOOLEAN ←TRUE,
  readOnly: BOOLEAN ←FALSE,
  commandProc: CommandProc,
  label: CommandItemLabel,
  clientData: LONG POINTER ←NIL];
```

```
CommandItemLabel: TYPE = RECORD [
  var: SELECT type: CommandItemLabelType FROM
    string => [string: XString.ReaderBody],
    bitmap => [bitmap: Bitmap],
    ENDCASE];
```

```
CommandItemLabelType: TYPE = {string, bitmap};
```

NewMakeCommandItem is just like **MakeCommandItem**, but allows the label to be a bitmap. If **label** is the **bitmap** variant, the client must leave the storage for the bitmap allocated as long as the item exists. **NewMakeCommandItem** is defined in FormWindowExtra3.mesa.

```
CommandProc: TYPE = PROCEDURE [
  window: Window.Handle,
  item:ItemKey, clientData: LONG POINTER];
```

A **CommandProc** is supplied by the client to **MakeCommandItem**. It is called whenever the user selects the command item. **window** is the **FormWindow** that the item is in. **item** is the key of the command item to which this **CommandProc** is attached.

29.2.2.4 Tagonly items

```
MakeTagOnlyItem: PROCEDURE [
  window: Window.Handle,
  myKey: ItemKey,
  tag: XString.Reader,
  visibility: Visibility ←visible];
```

Creates a **tagonly** item. **Tagonly** items are displayed as uneditable, nonselectable text.May raise Error[notAFormWindow, duplicateItemKey].

29.2.2.5 Text and Number Items

```
MakeTextItem: PROCEDURE [
  window: Window.Handle,
  myKey: ItemKey,
   tag: XString.Reader ← NIL,
   suffix: XString.Reader ← NIL,
   visibility: Visibility ← visible,
   boxed: BOOLEAN ← TRUE,
   readOnly: BOOLEAN ← FALSE,
   width: CARDINAL, -- in screen dots
   initString: XString.Reader ← NIL,
   wrapUnderTag: BOOLEAN ← FALSE,
   passwordFeedback: BOOLEAN ← FALSE,
   hintsProc: TextHintsProc ← NIL,
   nextOutOfProc: NextOutOfProc ← NIL,
   SPECIALKeyboard: BlackKeys.Keyboard ← NIL];
```

Creates a text item. Text items are user-editable text strings. The value of a text item is of type **XString.ReaderBody**. The user may select text, extend the selection, insert text, delete text, move and copy text, etc. Text items are fixed width but may grow and shrink vertically as the user enters and deletes text. A text item will contain nonattributed text only. **FormWindow** handles all storage allocation for the backing string.

width is the number of screen dots wide that the item should be. The item may grow arbitrarily long as the user enters text, but it will always retain the same width.

initString is the initial string to place in the text item. The bytes are copied by **FormWindow**.

wrapUnderTag specifies whether any text wider than the width of the text item should appear underneath the tag (**wrapUnderTag = TRUE**) or start at the left edge of the text item (**wrapUnderTag = FALSE**). Note: This feature is not yet implemented; that is, items always behave with **wrapUnderTag = FALSE**.

passwordFeedback indicates that the text should be displayed in an unreadable form (e.g. asterisks) rather than as normal characters. The correct value of the string is maintained internally, so that a call to **GetTextItemValue** will return the proper value. If any part of a passwordFeedback field is copied or moved, the underlying string is NOT copied.

If **hintsProc** is supplied, it is called to make a list of strings to be displayed to the user as a popup hint menu. (See **TextHintsProc** below.)

If **nextOutOfProc** is supplied, it is called when the user presses the NEXT key while the input focus is in this text item. This gives the client an opportunity to create more text items. After calling the **nextOutOfProc** or if no **nextOutOfProc** is supplied, the NEXT key causes the selection and input focus to move to the next text or window item in the form. See **NEXT key** in this chapter for further explanation.

If **SPECIALKeyboard** is supplied, it allows clients to make a special keyboard available to the user when typing into a text or number field.

May raise **Error[notAFormWindow, duplicateItemKey]**.

```
MakeDecimalItem: PROCEDURE [
   window: Window.Handle,
   myKey: ItemKey,
   tag: XString.Reader ← NIL,
   suffix: XString.Reader ← NIL,
   visibility: Visibility ← visible,
   boxed: BOOLEAN ← TRUE,
   readOnly: BOOLEAN ← FALSE,
   signed: BOOLEAN ← FALSE,
   width: CARDINAL, -- in screen dots --
   initDecimal: XLReal.Number ← XLReal.zero,
   wrapUnderTag: BOOLEAN ← FALSE,
   hintsProc: TextHintsProc ← NIL,
   nextOutOfProc: NextOutOfProc ← NIL,
   displayTemplate: XString.Reader ← NIL,
   SPECIALKeyboard: BlackKeys.Keyboard ← NIL];
```

Creates a decimal item. A decimal item is a text item that has a value of type XLReal.Number. (See MakeTextItem above for details of text items.) The user can type any text into the decimal item, but when the client calls GetDecimalItemValue to retrieve the value, FormWindow converts the string to XLReal.Number. initDecimal is the initial decimal value to place in the item. displayTemplate parameter is defined as in the XLReal.PictureReal. XLReal.PictureReal is used to display the value of the decimal item. The client may provide a keyboard interpretation with the SPECIALKeyboard parameter (see BlackKeys chapter). May raise Error[notAFormWindow, duplicateItemKey].

```
MakeIntegerItem: PROCEDURE [
   window: Window.Handle,
   myKey: ItemKey,
   tag: XString.Reader ← NIL,
   suffix: XString.Reader ← NIL,
   visibility: Visibility ← visible,
   boxed: BOOLEAN ← TRUE,
   readOnly: BOOLEAN ← FALSE,
   signed: BOOLEAN ← FALSE,
   width: CARDINAL, -- in screen dots --
   initInteger: LONG INTEGER ← 0,
   wrapUnderTag: BOOLEAN ← FALSE,
   hintsProc: TextHintsProc ← NIL,
   nextOutOfProc: NextOutOfProc ← NIL,
   SPECIALKeyboard: BlackKeys.Keyboard ← NIL ];
```

Creates an integer item. An integer item is a text item that has a value of type LONG INTEGER. (See MakeTextItem above for details of text items.) The user can type any text into the integer item, but when the client calls GetIntegerItemValue to retrieve the value, FormWindow converts the string to a LONG INTEGER. initInteger is the initial number to place in the item. The client may provide a keyboard interpretation with the SPECIALKeyboard parameter (see BlackKeys chapter). May raise Error[notAFormWindow, duplicateItemKey].

```
TextHintAction: TYPE = {replace, append, nil};

TextHintsProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [
   hints: LONG DESCRIPTOR FOR ARRAY OF XString.ReaderBody,
   freeHints: FreeTextHintsProc,
   hintAction: TextHintAction ← replace];

FreeTextHintsProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   hints: LONG DESCRIPTOR FOR ARRAY OF XString.ReaderBody];
```

The client may provide a **TextHintsProc** to **MakeTextItem**, **MakeDecimalItem**, and **MakeIntegerItem**. Whenever the user points at the mouse menu for a text item, this procedure is called and the hints returned are used to construct a popup menu that is displayed.

When the user selects one of the strings from the popup menu, one of three things will happen, depending on the **hintAction** returned by the **TextHintsProc**. If **hintAction = replace**, the selected string will replace the current value of the text item. If **hintAction = append**, the selected string will be appended to the current value of the text item. If **hintAction = nil**, the current value of the text item will not change. **hintAction = nil** is useful for displaying "help-like" information to the user for text items that do not have a finite number of possible values, such as a file name.

freeHints is a procedure that will be called after the hint menu has been taken down to allow the client to free any storage that was allocated when creating the hints array.

29.2.2.6 Window Items

```
MakeWindowItem: PROCEDURE [
   window: Window.Handle,
   myKey: ItemKey,
   tag: XString.Reader ← NIL,
   visibility: Visibility ← visible,
   boxed: BOOLEAN ← TRUE,
   size: Window.Dims,
   nextIntoProc: NextIntoProc ← NIL]
   RETURNS [clientWindow: Window.Handle];
```

Creates a window item. A window item is a window (**Window.Handle**) that is a child of the **FormWindow** and can contain anything the client desires. A window with dimensions **size** is created and returned as **clientWindow**. It is expected that the client will associate a display proc (see **Window.SetDisplayProc**) and a **TIP.NotifyProc** with the window. The window may be treated just like any other window, *except* **FormWindow.SetWindowItemSize** *must* be used to change the size of the window rather than calling **Window.SlideAndSize** directly. This allows **FormWindow** to move any other items, if necessary, to accomodate the different-sized window item.

If **nextIntoProc** is supplied, it is called when the user presses the NEXT key in an item just before this window item. This gives the window item an opportunity to gain control of the NEXT key by setting the input focus to be the window item's window. The window item may then retain control of the NEXT key within the window item. When the window item no longer wants to process the NEXT key (for instance, when the NEXT key should move the selection outside the window item), the window item client must call **FormWindow.TakeNEXTKey**, which returns the NEXT key processing to the form window. (See §29.2.10 for an explanation of the NEXT key.)

May raise Error[notAFormWindow, duplicateItemKey].

```
SetWindowItemSize: PROCEDURE [
   window: Window.Handle,
   windowItemKey: ItemKey,
   newSize: Window.Dims];
```

```
SetWindowItemSizeExtra: PROCEDURE [
   window: Window.Handle,
   windowItemKey: ItemKey,
   newSize: Window.Dims,
   repaint: BOOLEAN ← TRUE];
```

SetWindowItemSize (or **SetWindowItemSizeExtra**) should be used to change the size of a window item's window. The client should *never* call **Window.SlideAndSize** directly. Any items below the window item are moved down or up to accommodate the new dimensions. **window** is the form window that the window item is in. **windowItemKey** must be the key of a window item. **newSize** indicates the new dimensions. **SetWindowItemSizeExtra** is defined in FormWindowExtra.mesa. May raise **Error[notAFormWindow, invalidItemKey, wrongItemType]**.

29.2.2.7 Destroying Items

```
DestroyItem: PROCEDURE [
   window:Window.Handle,
   item: ItemKey,
   repaint: BOOLEAN ← TRUE];
```

DestroyItem destroys **item**. Most clients will not need to use this procedure, since **FormWindow.Destroy** destroys all the items in the **FormWindow**. May raise **Error[notAFormWindow, invalidItemKey]**.

```
DestroyItems: PROCEDURE [
   window:Window.Handle,
   item: LONG DESCRIPTOR FOR ARRAY OF ItemKey,
   repaint: BOOLEAN ← TRUE];
```

DestroyItems destroys several items at once. Most clients will not need to use this procedure, since **FormWindow.Destroy** destroys all the items in the **FormWindow**. May raise **Error[notAFormWindow, invalidItemKey]**.

29.2.3 Getting and Setting Values

The client may examine or change the value of an item. All **GetXXXItem** procedures return the current value of an item. All **SetXXXItem** procedures take a given new value and change the value internally, as well as updating the screen if necessary.

In all these procedures, **window** is the **FormWindow** the item is in. **item** uniquely identifies the item to get/set the value of.

Note: There are two ways to get the value of a text item. **GetTextItemValue** copies the bytes of the string so that the storage for the returned value is owned by the client.

LookAtTextItemValue simply returns a pointer to the **FormWindow**-owned backing string. This value is therefore read-only and must be released when the client is done examining it by calling **DoneLookingAtTextItemValue**.

All of these may raise **Error[notAFormWindow, invalidItemKey, wrongItemType]**. If the item is in a neutral state (see § 29.2.8), these will raise **ItemError[neutralItem]**.

29.2.3.1 Getting Values

```
GetBooleanItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [value: BOOLEAN];

GetChoiceItemValue: PROCEDURE [
   window:Window.Handle,
   item: ItemKey]
   RETURNS [value: ChoiceIndex];

GetDecimalItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [value:XLReal.Number];
```

May raise **XLReal.Error [notANumber]**.

```
GetIntegerItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [value: LONG INTEGER];
```

May raise **XString.InvalidNumber** or **XString.Overflow**.

```
GetMultipleChoiceItemValue: PROCEDURE [
   window:Window.Handle,
   item: ItemKey, zone: UNCOUNTED ZONE]
   RETURNS [values: LONG DESCRIPTOR FOR ARRAY OF ChoiceIndex];
```

The **zone** parameter is added. The storage for the **DESCRIPTOR** will be allocated out of **zone** and the storage must be freed by the client.

```
GetTextItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   zone: UNCOUNTED ZONE]
   RETURNS [value: XString.ReaderBody];
```

GetTextItemValue copies the string. Storage for the bytes is allocated out of **zone**. The client should free the storage using **XString.FreeReaderBytes** and **zone**.

```
GetWindowItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [value: Window.Handle];

LookAtTextItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [value: XString.ReaderBody];

DoneLookingAtTextItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey];
```

LookAtTextItemValue does not copy the string but returns a pointer to it. **value** should *not* be changed by the client. Clients using **LookAtTextItemValue** must call **DoneLookingAtTextItemValue** when done examining it. During the time between these calls, if another client calls **LookAtTextItemValue** or **SetTextItemValue** for the same text item, the second client's process will WAIT.

```
GetNextAvailableKey: PROCEDURE [window: Window.Handle]
   RETURNS [key: ItemKey];
```

Returns the next available item key: MAX[usedKeys] + 1.

29.2.3.2 Setting Values

All the **SetXXXItem** procedures take a **repaint: BOOLEAN**. If **repaint** = TRUE and the item is currently visible, it will be repainted with the new value. If **repaint** = FALSE, the item will not be repainted until **FormWindow.Repaint** is called. This allows the client to change the values of several items at once without the screen flashing for each item. **Warning:** After calling any procedure with **repaint** = FALSE, **FormWindow.Repaint** must be called. Otherwise, the screen will be inconsistent with the internal values.

Caution: If a change proc does a **SetXXXItemValue**, the client should take extreme care to prevent infinite recursion. (See §29.3.1.)

```
SetBooleanItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   newValue: BOOLEAN,
   repaint: BOOLEAN ←TRUE];

SetChoiceItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   newValue: ChoiceIndex,
   repaint: BOOLEAN ←TRUE];
```

May raise **FormWindow.Error[invalidChoiceNumber]**.

```
SetDecimalItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   newValue: XLReal.Number,
   repaint: BOOLEAN ←TRUE];

SetIntegerItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   newValue: LONG INTEGER,
   repaint: BOOLEAN ←TRUE];

SetMultipleChoiceItemValue: PROCEDURE [
   window:Window.Handle,
   item: ItemKey,
   newValue: LONG DESCRIPTOR FOR ARRAY OF ChoiceIndex,
   repaint: BOOLEAN ←TRUE];
```

May raise FormWindow.Error[invalidChoiceNumber].

```
SetTextItemValue: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   newValue: XString.Reader,
   repaint: BOOLEAN ←TRUE];
```

29.2.4 "Changed" BOOLEAN

Every item that has a value that the user can change (all except tagonly and command items) has a "changed" boolean associated with it. All items are created with this boolean set to FALSE. FormWindow automatically sets this boolean to TRUE whenever the user changes the item or when a client calls one of the SetXXXItemValue procedures. This allows the client to determine which items have changed when, for example, the user selects "Done" or "Apply" on a property sheet. The client is responsible for resetting the changed boolean to false by calling ResetChanged or ResetAllChanged after examining the changed boolean with HasBeenChanged or HasAnyBeenChanged.

```
HasAnyBeenChanged: PROCEDURE [
   window: Window.Handle]
   RETURNS [yes: BOOLEAN];
```

HasAnyBeenChanged returns true if any item's changed boolean is TRUE. May raise Error[notAFormWindow].

```
HasBeenChanged: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [yes: BOOLEAN];
```

HasBeenChanged returns TRUE if the user has changed item. The client may reset the changed boolean to FALSE by using ResetChanged or ResetAllChanged. If item is tagonly or command, HasBeenChanged returns FALSE. May raise Error[notAFormWindow, invalidItemKey].

```
ResetChanged: PROCEDURE [window: Window.Handle, item: ItemKey];
```

ResetChanged sets the changed boolean of **item** to FALSE. May raise Error[notAFormWindow, invalidItemKey].

```
ResetAllChanged: PROCEDURE [window: Window.Handle];
```

ResetAllChanged sets the changed boolean of all items to FALSE. May raise Error[notAFormWindow].

```
SetChanged: PROCEDURE [
   window: Window.Handle,
   item: ItemKey];
```

SetChanged sets the changed boolean of **item** to TRUE. May raise Error[notAFormWindow, invalidItemKey].

```
SetAllChanged: PROCEDURE [
   window: Window.Handle];
```

SetAllChanged sets the changed boolean of all items to TRUE. May raise Error[notAFormWindow].

29.2.5 Visibility

```
Visibility: TYPE = {visible, invisible, invisibleGhost};
```

An item either is or is not displayed in the form window. If an item is displayed in the form window, it is **visible**. If an item is not currently displayed, it is either **invisible** or **invisibleGhost**. If it is **invisible**, it does not take up any space on the screen; any items below it move up to take its screen space. If an item is **invisibleGhost**, the space that it would occupy were it visible is white on the screen. An item's visibility can be changed anytime by calling **SetVisibility**.

```
GetVisibility: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [visibility: Visibility];
```

GetVisibility returns the current visibility of **item**. May raise Error[notAFormWindow, invalidItemKey].

```
SetVisibility: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   visibility: Visibility,
   repaint: BOOLEAN ←TRUE];
```

SetVisibility sets the visibility of **item**. If **repaint** = TRUE and the item's visibility is changing, the form window will be repainted. If **repaint** = FALSE, the form window will not be repainted until FormWindow.Repaint is called. This allows the client to change the visibility of several items at once without the screen flashing for each item. **Warning:**

After calling **SetVisibility** with **repaint** = FALSE, **FormWindow.Repaint** must be called. Otherwise, the screen will be inconsistent with internal values. May raise **Error[notAFormWindow, invalidItemKey]**.

29.2.6 Layout

The exact layout of items in a form window is done by calling various procedures specified below, after creating the items to be laid out. If an item is not explicitly laid out, it will not appear in the form window at all. Note that **FormWindow.DefaultLayout** may be used when the client is not concerned with the exact placement of items, but wants a functional form window.

There are two different types of layout. The most common is flexible layout, which allows text, decimal, integer, and window items to grow and shrink (and all other items are moved around accordingly) as the user or client changes their values. Flexible layout is done by calling such procedures as **AppendLine** and **AppendItem**. The other is fixed layout, which allows the client to specify exactly where items will go by calling **SetItemBox**, but does not allow text, decimal, integer, and window items to grow or shrink. All items stay where they are laid out unless the client calls **SetItemBox** again.

29.2.6.1 Flexible Layout

A form window with flexible layout consists of horizontal lines with zero or more items on each line. Lines now are always just tall enough to hold the items on that line. The **spaceAboveLine** parameter specifies the amount of white spae to leave above each line. Any desired horizontal spacing may be accomplished by using appropriate margins between items. Items may be lined up horizontally by using **TabStops** (see §29.2.6.2 below).

Lines are created by calling **AppendLine** or **InsertLine**. Items are placed on a line by calling **AppendItem** or **InsertItem**. The Append routines are used to add items after the previously created line or item. The Insert routines are used to add items between previously created items or lines.

```
AppendLine: PROCEDURE [
   window: Window.Handle,
   spaceAboveLine: CARDINAL ← 0]
   RETURNS [line: Line];
```

AppendLine creates a new line and appends it to the bottom of the form window. All items must be placed on a line, so **AppendLine** must be called before any calls to **AppendItem**. The line returned by **AppendLine** should be passed to **AppendItem** or **InsertItem**. **window** is the **FormWindow** the line is being appended to. May raise **Error[notAFormWindow]**.

```
Line: TYPE;
```

Line uniquely identifies a line and is returned by **AppendLine** and **InsertLine**. A **Line** must be passed to **AppendItem** and **InsertItem**.

```
AppendItem: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   line: Line,
    preMargin: CARDINAL←0,
    tabStop: CARDINAL←nextTabStop,
    repaint: BOOLEAN←TRUE];
```

AppendItem appends **item** to **line**.

preMargin is the number of pixels of white space to place before the left edge of this item. If tabs have been set, **preMargin** is added after placing the item at its tab stop.

tabStop is the ordinal number of the tab stop at which to place this item. If the default is taken, the next tab stop on the line after the previous item is used. If no tabs have been defined (i.e., **SetTabStops** has never been called), **tabStop** is ignored. See §29.2.6.2 for more detail on tabs.

repaint specifies whether the screen should be repainted after the **AppendItem** is done. When called from the client's **LayoutProc**, **repaint** is ignored and the items are not painted until the **LayoutProc** returns. When not called from the client's **LayoutProc**, and **repaint** = TRUE, the form window will be repainted immediately after appending the item. When not called from the client's **LayoutProc**, and **repaint** = FALSE, the form window will not be repainted until **FormWindow.Repaint** is called. This allows the client to add several items at once without the screen flashing for each new item. **Warning:** After calling **AppendItem** with **repaint** = FALSE, **FormWindow.Repaint** must be called. Otherwise, the screen will be inconsistent with internal values.

May raise **Error[notAFormWindow, invalidItemKey, noSuchLine]**.

```
InsertLine: PROCEDURE [
    window: Window.Handle,
    before: Line,
    spaceAboveLine: CARDINAL←0]
    RETURNS [line: Line];
```

InsertLine inserts a new line before (above) an existing line. The **spaceAboveLine** parameter indicates how much space (in screen dots) to leave between the previous line and this line. This allows clients to leave white space at the top of the form before the first line and also provides an easy way to put white space in a form. (See **AppendLine** for details of creating a line.) May raise **Error[notAFormWindow, noSuchLine]**.

```
InsertItem: PROCEDURE [
    window: Window.Handle,
    item: ItemKey,
    line: Line,
    beforeItem: ItemKey,
    preMargin: CARDINAL←0,
    tabStop: CARDINAL←nextTabStop,
    repaint: BOOLEAN←TRUE];
```

InsertItem inserts item to the left of **beforeItem** on **line**. See **AppendItem** for details of placing an item on a line. May raise **Error [notAFormWindow, invalidItemKey, noSuchLine, itemNotOnLine]**.

```
RemoveItemFromLine: PROCEDURE [
    window: Window.Handle,
    item: ItemKey,
   line: Line,
   repaint: BOOLEAN ← TRUE];
```

RemoveItemFromLine will "unlayout" an item that has been previously laid out. This allows clients to move an item from one place on the form to another without destroying and recreating the item, by calling **RemoveItemFromLine** followed by a call to **AppendItem** or **InsertItem**. **RemoveItemFromLine** will not destroy the item. The item will be "in limbo" until it is laid out again using **AppendItem** or **InsertItem**.

```
LayoutInfoFromItem: PROCEDURE [
   window: Window.Handle, item: ItemKey]
   RETURNS [ line: Line, margin: CARDINAL,
   tabStop: CARDINAL, box: Window.Box];
```

LayoutInfoFromItem returns various layout characteristics of **item**. May raise **Error[notAFormWindow, invalidItemKey]**.

```
LineUpBoxes: PROCEDURE [window: Window.Handle,
   items: LONG DESCRIPTOR FOR ARRAY OF ItemKey ← NIL];
```

Calling this procedure will force the boxes of the specified items to line up vertically, as in most ViewPoint property sheets. If no items are specified, and a fixed-pitch font is used, the first item on every line will line up as shown in Figure 29.1.

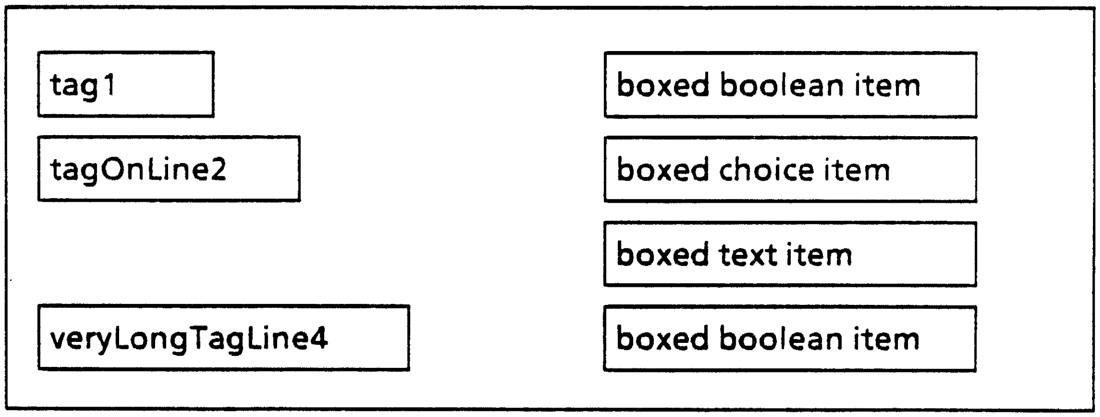


Figure 29.1 **LineUpBoxes**

The specified items must be the first item on their line. The longest tag is measured; then the boxed part of each item appears at the next available tab stop after the longest tag. This also works for non-boxed items.

29.2.6.2 Tabs

```
TabType: TYPE = {fixed, vary};

TabStops: TYPE = RECORD [
   variant: SELECT type: TabType FROM
   fixed => [interval: CARDINAL],
vary = > [list: LONG DESCRIPTOR FOR ARRAY OF CARDINAL]
ENDCASE ];
```

The client may specify tab stops to facilitate lining up items one directly below the other. Tabs may be specified two ways: fixed and varying. Fixed tab stops are specified by a single CARDINAL (**interval**) that indicates a tab stop at each **interval** pixel, such as if **interval** = 10, there will be tab stops at 10, 20, 30, etc. Varying tab stops are specified by an ARRAY OF CARDINAL, each element of the ARRAY indicating the number of pixels from the left edge of the window. Typically, a client will call **SetTabStops** at the beginning of the **LayoutProc**, then call **AppendLine** and **AppendItem** repeatedly, taking the **nextTabStop** default for each item.

```
noTabStop: CARDINAL = CARDINAL.LAST-1;
```

Can be used with **AppendItem** and **InsertItem** to indicate that this item should ignore tab stops completely.

```
defaultTabStops: TabStops = [fixed[interval: 100]];
```

```
SetTabStops: PROCEDURE [window: Window.Handle, tabStops: TabStops];
```

SetTabStops sets the tab stops for **window**. Any items laid out before to this call will now be moved to conform to these tab stops. May raise **Error[notAFormWindow]**.

```
nextTabStop: CARDINAL = ...;
```

Used for item layout, it is the default for the **tabStop** parameter for **AppendItem** and **InsertItem**. Indicates that the next item should be placed at the next tab stop.

```
GetTabStops: PROCEDURE [window: Window.Handle]
  RETURNS [tabStops: TabStops];
```

GetTabStops returns the current tab stops for **window**. If no tab stops have been set for **window**, **tabStops** will be fixed with an interval of 0. May raise **Error[notAFormWindow]**.

29.2.6.3 Fixed Layout

```
SetItemBox: PROCEDURE [
  window: Window.Handle,
  item: ItemKey,
  box: Window.Box];
```

SetItemBox is used to set the exact position of an item for fixed layout. With fixed layout, all items stay right where they are laid out unless the client calls **SetItemBox** again. With fixed layout, text, decimal, integer, and window items will not grow or shrink. **SetItemBox** is incompatible with flexible layout (such as. **AppendLine, AppendItem, SetTabStops**, etc). **Note:** Either all layout must be flexible, or all layout must be fixed. Attempting to mix them will raise **Error[notAFormWindow, invalidItemKey]**.

29.2.7 Save and Restore

Restore: PROCEDURE [window: Window.Handle];

Save: PROCEDURE [window: Window.Handle];

Restore and **Save** deal with restoration of a form window to a previous state. **Save** causes the current item values to be saved. **Restore** causes the previously saved values to be copied back into the form. A **Restore** done before a **Save** is a no-op. **Save** done after **Save** (but before a **Restore**) overwrites the first **Save**. These procedures support the Reset function of property sheets. May raise **Error[notAFormWindow]**.

29.2.8 Neutral Properties

There are times when the client wants to indicate to the user that a particular item no longer has a meaningful value. One way of achieving this is to remove the item completely from the display by setting its visibility. If the client wishes the item to remain visible, however, a different mechanism can be used. All items (except Command items) can be placed into a *neutral* state, indicating that they no longer have a specific value. When an item becomes neutral, it's display is changed such that any value indications are removed and uniform diagonal gray stripes are painted over the item's tag, suffix, and value box. Setting an item to or from the neutral state is done by calling **SetItemNeutralness**.

Items will return to a normal state whenever the user selects a value for that item. In the case of text, integer, and decimal items, the item will return to a normal state whenever the item receives the text input focus. This can be done either by the user POINTing up inside the item value box or by NEXTing into the item. In addition, an item will return to a normal state whenever a client calls **SetXXXItemValue**.

When a text or number item becomes neutral, it's value is cleared before the gray stripes are painted over the value box. For boolean and choice items, the value boxes are dehighlighted before the stripes are painted. When boolean and choice items are returned to a normal state, the values they had prior to changing to neutral are restored. Text and number items, however, lose their previous values and are returned to a normal state with their values fields empty.

When an item is in the neutral state, it actually has no value at all. Because of this, calling **GetXXXItemValue** for a neutral item will raise **ItemError[neutralItem]**. In addition, since the neutral state is not defined for command items, calling **SetItemNeutralness** for a command item will raise **Error[wrongItemType]**.

```
SetItemNeutralness: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   neutral: BOOLEAN,
   repaint: BOOLEAN ← TRUE];
```

The **neutral** parameter is used to indicate whether the item should become neutral, or return to a normal state and appearance. **window** is the **FormWindow** containing the item, and **item** is the specific **ItemKey** uniquely identifying the item. **repaint** indicates whether the client wishes the display to be updated. If this value is FALSE, the item will not change appearance, but the internal state of the item will be changed. The next time the FormWindow is repainted, the display will be changed to reflect the current state of the item.

```
NeutralDisplayProc: PROCEDURE [
   window: Window.Handle,
   box: Window.Box ← Window.nullBox];
```

When a window item is set to neutral, only the item's tag and suffix will be painted with gray stripes. Painting the interior of the window is the responsibility of the client. For the convenience of clients who wish the item to have a uniform neutral appearance, **NeutralDisplayProc** will display the neutral gray stripes in **box** for a given window. If the default value for **box** is used, then stripes will be painted over the entire window.

```
IsNeutral: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [yes: BOOLEAN];
```

IsNeutral indicates whether a given **item** is in the neutral state. Will raise **Error[wrongItemType]** if the item is a command item.

All of the procedures listed in this section and **ItemError** are defined in **FormWindowExtra5**.

29.2.9 Item Popup Menus

```
SetPopupProc: PROCEDURE [
   window: Window.Handle,
   proc: PopupProc]
   RETURNS [ old: PopupProc];

PopupProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item: FormWindow.ItemKey]
   RETURNS [ menu: MenuData.MenuHandle ← NIL,
   freeProc: MenuFreeProc ← NIL];

MenuFreeProc: TYPE = PROCEDURE [
   window: Window.Handle,
   menuHandle: MenuData.MenuHandle];
```

A **PopupProc** provides the client with a mechanism for attaching a popup menu to any item in a **FormWindow**. If a **PopupProc** exists for a **FormWindow**, it will be called whenever the user CHORDs over any **item** in the **window**. If **menu** is non NIL, the popup menu described by it will be put up. **freeProc** is a procedure which will be called when the popup menu is taken down so that the client can free any storage allocated for the MenuHandle. The **PopupProc** is set for a **FormWindow** by calling **SetPopupProc**. If a **PopupProc** does not exists for a **FormWindow**, the CHORDing action will be ignored. **SetPopupProc** may raise **Error[notAFormWindow]**.

If the **PopupProc** returns a NIL value for **menu**, the popup menu will not be put up and the CHORDing action will be ignored.

The sensitive area for the CHORDing action over an item is the entire area covered by the item's tag, suffix, and value box.

```
NeutralPopupProc: PopupProc;
```

NeutralPopupProc is provided for the convenience of those clients that wish to provide a popup menu for setting items to a neutral state. The popup menu for this **PopupProc** consists of a single command which will be used to set that item to a neutral state. This procedure will return a **MenuHandle** and **MenuFreeProc** for text, integer, decimal, boolean, choice, and window items. A popup menu is not provided for readOnly, tag, or command items.

All of these types and procedures are defined in **FormWindowExtra5**.

29.2.10 Miscellaneous TYPEs

```
Bitmap: TYPE = RECORD[
   height, width: CARDINAL,
   bitsPerLine: CARDINAL,
   bits: Environment.BitAddress];
```

A **Bitmap** is the data structure that is passed to **MakeBooleanItem** and **MakeChoiceItem** for items that are to be displayed as bitmaps. **height** is the height in pixels, of the bitmap. **width** is the width in pixels, of the bitmap. **bits** is a pointer to the actual bits in the bitmap. **bitsPerLine** is the number of bits in each line of bits. **bitsPerLine** is usually greater than or equal to **width**, and is often a multiple of 16.

```
ChangeReason: TYPE = {user, client, restore};
```

A **ChangeReason** is passed to a **GlobalChangeProc**, **BooleanChangeProc**, and **ChoiceChangeProc**. It indicates whether the change was caused by the user, or by the client calling **SetXXXItemValue**, or by the client calling **Restore**.

29.2.11 Miscellaneous Item Operations

```
GetReadOnly: PROCEDURE [window: Window.Handle, item: ItemKey]
   RETURNS [readOnly: BOOLEAN];
```

GetReadOnly returns the current value of the **readOnly** BOOLEAN for item. May raise **Error[notAFormWindow, invalidItemKey]**.

```
GetTag: PROCEDURE [
   window: Window.Handle,
item: ItemKey]
RETURNS [tag: XString.Reader];
```

GetTag returns the tag associated with **item**. May raise **Error[notAFormWindow, invalidItemKey]**.

```
SetSelection: PROCEDURE [
  window: Window.Handle,
  item: ItemKey,
  firstChar: CARDINAL ← 0,
  lastChar: CARDINAL ← CARDINAL.LAST];
```

SetSelection sets the current selection to be **item**. This is useful for helping the user correct an incorrect user entry. **item** must be a text, decimal, or integer item. **firstChar** is the first character of the portion of the string to be selected and highlighted. **lastChar** is the last character of the portion of the string to be selected and highlighted. The defaults for **firstChar** and **lastChar** causes the entire string to be selected. May raise **Error[notAFormWindow, invalidItemKey, wrongItemType]**.

```
SetInputFocus: PROCEDURE [
  window: Window.Handle,
  item: ItemKey,
  beforeChar: CARDINAL ← CARDINAL.LAST];
```

SetInputFocus sets the current input focus to be in **item**. This is useful for highlighting an incorrect user entry. item must be a text, decimal, or integer item. **beforeChar** is the character before which the input focus should go. The default causes the input focus to be at the end of the string. May raise **Error[notAFormWindow, invalidItemKey, wrongItemType]**.

```
SetReadOnly: PROCEDURE [
  window: Window.Handle,
  item: ItemKey,
  readOnly: BOOLEAN]
  RETURNS [old: BOOLEAN];
```

SetReadOnly sets the current "readOnly-ness" of item and returns the old value. May raise **Error[notAFormWindow, invalidItemKey]**.

```
SetItemWidth: PROCEDURE [window: Window.Handle, item: ItemKey,
  width: CARDINAL];
```

This procedure sets the width of an item. Normally, items are as wide as they need to be to display the text of the item (except text, decimal, and integer items whose width is specified when the items are created). **SetItemWidth** overrides the normal width of the item and thus could result in the text of the item being truncated. **SetItemWidth** should therefore be used with great caution. In particular, programmers should keep in mind that applications are intended to be multinational and strings in other languages are often longer than their English equivalents. This layout procedure can only be used with a flexible layout.

```
SetItemFont: PROCEDURE [
  window: Window.Handle,
  item: ItemKey,
  newFont: SimpleTextFont.MappedFontHandle ← NIL,
  repaint: BOOLEAN ← TRUE];
```

SetItemFont changes the font of a text or number item. It does not affect the tag or suffix. If **newFont** is **NIL**, the system font is used. May raise **Error[notAFormWindow, invalidItemKey]**. **SetItemFont** is defined in FormWindowExtra4.mesa.

```
SetLosingFocusProc: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   proc: LosingFocusProc];
```

```
LosingFocusProc: TYPE = PROCEDURE [window: Window.Handle, item: ItemKey];
```

SetLosingFocusProc associates **proc** with **item**. **proc** is called whenever **item** loses the input focus, allowing clients to undo things that were done when the input focus was set, e.g. clear some SoftKeys. **item** must be a text or number item. May raise **Error[notAFormWindow, invalidItemKey]**. **SetLosingFocusProc** and **LosingFocusProc** are defined in FormWindowExtra6.mesa.

```
SetTookFocusProc: PROCEDURE [window: Window.Handle, proc: LosingFocusProc];
```

```
TookFocusProc: TYPE = PROCEDURE [window: Window.Handle, item: ItemKey];
```

SetTookFocusProc associates **proc** with **window**. **proc** is called whenever the form window takes the input focus. **item** is the item that took the input focus, and will be a text or number item. May raise **Error[notAFormWindow, invalidItemKey]**. **SetTookFocusProc** and **TookFocusProc** are defined in FormWindowExtra6.mesa.

29.2.12 NEXT Key

When the user presses the NEXT key while the input focus is in a form window (more exactly: in a text, decimal, or integer item in a form window), the form window does the following:

1. If the item with the input focus has a **NextOutOfProc**,it is called. This gives the client an opportunity to, for example, add another blank text item after this one.

2. Find the next text, decimal, integer, or window item. Note: If the client added another text item after the one that had the input focus, that new item will be the one found by form window.

3a. If the next item is a text, decimal, or integer item, the input focus and selection are moved to that item.

3b. If the next item is a window item and the window item has a **NextIntoProc**, it is called, giving the window item an opportunity to take the input focus. For example, if the window item contains a table of values, the NEXT key could be used to step from entry to entry through the table, but the window item's **TIP.NotifyProc** would have to do this. **Note:** If a **NextIntoProc** is supplied for a window item, it MUST call **TIP.SetInputFocus** so that all further NEXT key notifications will go to the window item. When the window item no longer wants the NEXT key (such as the user has NEXTed out of the last entry of the table), it must call **FormWindow.TakeNEXTKey**. **TakeNEXTKey** proceeds as in steps 2 and 3.

3c. If the next item is a window item, but the window item does not have a **NextIntoProc**, the form window repeats steps 2 and 3.

```
NextIntoProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item: ItemKey];
```

A **NextIntoProc** can be provided by the client with window items. If provided, the **NextIntoProc** will be called when the user NEXTs into the item using the NEXT key. (See the discussion above.)

```
NextOutOfProc: TYPE = PROCEDURE [
   window: Window.Handle,
   item: ItemKey];
```

A **NextOutOfProc** can be provided by the client with text, decimal, and integer items. If provided, the **NextOutOfProc** is called when the user hits the NEXT key while the input focus is in an item just before this one. See the discussion above.

```
SetNextOutOfProc: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   nextOutOfProc: NextOutOfProc]
   RETURNS [old: NextOutOfProc];
```

SetNextOutOfProc sets the **NextOutOfProc** for a text, decimal, or integer item. This is useful when the **NextOutOfProc** for a text item creates another text item after itself. After creating the new item, the client will probably want to set the **NextOutOfProc** for the old item to NIL, so that next time the user NEXTs out of the old item, the selection and input focus will simply move to the new item rather than creating yet another new item.

```
GetNextOutOfProc: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [NextOutOfProc];
```

GetNextOutOfProc returns the **NextOutOfProc** for **item**.

```
TakeNEXTKey: PROCEDURE [
   window: Window.Handle,
   item: ItemKey];
```

TakeNEXTKey informs form window that the window item which was handling the NEXT item is done with it and the input focus should be passed on to the next item that can take it. **item** identifies the window item that is involved. May raise **Error[notAFormWindow, wrongItemType]**.

29.2.13 SIGNALs and ERRORs

```
Error: ERROR [code: ErrorCode];
```

ErrorCode: TYPE = MACHINE DEPENDENT {notAFormWindow(0), wrongItemType, invalidChoiceNumber, noSuchLine, alreadyAFormWindow, invalidItemKey, itemNotOnLine, duplicateItemKey, incompatibleLayout, alreadyLaidOut, last(15)};

notAFormWindow	The term notAFormWindow means the window passed in to the procedure is not a form window. Any FormWindow procedure, except Create and IsIt, may raise this error.
wrongItemType	The term wrongItemType means the item passed in to the FormWindow procedure is the wrong type. For example, GetChoiceItemValue must be passed a choice item.
invalidChoiceNumber	The term invalidChoiceNumber means the choice number supplied does not match any of the choice numbers in the ChoiceItems.
noSuchLine	The term noSuchLine means the line supplied to AppendItem or InsertItem was not previously created.
alreadyAFormWindow	The term alreadyAFormWindow means the window passed in is already a form window. Raised if a FormWindow is passed into Create.
invalidItemKey	The term invalidItemKey means an ItemKey was used for which there was no item created.
itemNotOnLine	The term itemNotOnLine means an attempt was made to insert an item on a line before an item that is not on that line. See InsertItem.
duplicateItemKey	The term duplicateItemKey means an item was created with the key of another item. ItemKeys must be unique.
incompatibleLayout	The term incompatibleLayout means the client is attempting to intermix fixed and flexible layout styles.
alreadyLaidOut	The term alreadyLaidOut means an attempt was made to specify the layout for an item more than once.

LayoutError: SIGNAL [code: LayoutErrorCode];

LayoutErrorCode: TYPE = {onTopOfAnotherItem, notEnufTabsDefined};

The following ERROR and ErrorCode are found in FormWindowExtra5.

ItemError: ERROR [errorCode: ErrorCode];

ErrorCode: TYPE = MACHINE DEPENDENT {neutralItem(0), mixedItem, last(15)};

neutralItem	The term neutralItem means that a call to GetXXXItemValue was made on an item in the neutral state.
mixedItem	mixedItem presently has no meaning.

29.2.14 Multinational items

```
Flushness: TYPE = SimpleTextDisplay.Flushness;

StreakSuccession: TYPE = SimpleTextDisplay.StreakSuccession;

GetFlushness: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [old: Flushness];

SetFlushness: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   new:Flushness]
   RETURNS [old:Flushness];

GetStreakSuccession: PROCEDURE [
   window: Window.Handle,
   item: ItemKey]
   RETURNS [old: StreakSuccession];

SetStreakSuccession: PROCEDURE [
   window: Window.Handle,
   item: ItemKey,
   new:StreakSuccession]
   RETURNS [old:StreakSuccession];
```

29.3 Usage/Examples

29.3.1 Calling ChangeProcs

There are three ways for a client to determine if an item has been changed. (1) The client may supply a **GlobalChangeProc** that governs the entire window, (2) it may supply a **XXXChangeProc** for certain items (such as choice and boolean), and (3) it may examine the "changed" boolean associated with each item.

An item can change because the user changes the item, or because a client calls **SetXXXItemValue**, or because a client calls **Restore**.

The two kinds of change procs are called whenever the "changed" boolean goes from false to true (whether that is caused by user actions or client actions). The following describes the exact order of events for each source of change:

- User action

  1. Change value of item and set "changed" boolean.
  2. Call local change proc, if any.
  3. Call global change proc, if any.

- Client call to SetXXXItemValue

  1. Change value of item and set "changed" boolean.
  2. Call local change proc, if any.
  3. Call global change proc, if any.

- Client call to **Restore**

  1. Change value of item and set "changed" boolean.
  2. Call global change proc, if any, with nullItemKey.

Note: If a change proc does a SetXXXItemValue, the client should take extreme care to prevent infinite recursion.

29.3.2 Creating a Simple FormWindow

```
MyItems: TYPE = {boolean, choice, text};



.
.
shell: StarWindowShell.Handle ← StarWindowShell.Create [...];
formWindow: Window.Handle ← StarWindowShell.CreateBody [shell];
FormWindow.Create [window: formWindow, makeItems: MakeItems,
   layoutProc: DoLayout];
.
.
.

MakeItems: FormWindow.MakeItemsProc = {
   <<[window: Window.Handle, clientData: LONG POINTER]>>
   tag: XString.ReaderBody;

   -- Make a boolean item
   BEGIN
   booleanLabel: FormWindow.BooleanItemLabel ← [string[
      XString.FromSTRING ["This is a boolean item"L]]];
   tag ← XString.FromSTRING ["Tag"L];
   FormWindow.MakeBooleanItem [
      window: window, myKey: MyItems.boolean.ORD,
      tag: @tag, label: booleanLabel,
      initBoolean: FALSE];
   END;

   -- Make a choice item
   BEGIN
   choice1: XString.ReaderBody ← XString.FromSTRING["Choice One"L];
   choice2: XString.ReaderBody ← XString.FromSTRING["Choice 2"L];
   choices: ARRAY [0..2) OF FormWindow.ChoiceItem ← [
      [ string[0, choice1] ],
      [ string[1, choice2] ] ];
   tag ← XString.FromSTRING ["Choice item"L];
FormWindow.MakeChoiceItem [
   window: window, myKey: MyItems.choice.ORD,
   tag: @tag, values: DESCRIPTOR[choices],
   initChoice: 0];
END;

-- Make a text item
tag ← XString.FromSTRING ["Text item"L];
FormWindow.MakeTextItem[
window: window, myKey: MyItems.text.ORD,
tag: @tag, width: 30];
};
```

```
DoLayout: FormWindow.LayoutProc = {
   <<[window: Window.Handle, clientData: LONG POINTER]>>

   FormWindow.SetTabStops [window: window, tabStops: [ fixed [100] ] ];

   line: FormWindow.Line ← FormWindow.AppendLine [window];

   -- Put boolean and choice item on line 1
   FormWindow.AppendItem[window, MyItems.boolean.ORD, line];
   FormWindow.AppendItem[window, MyItems.choice.ORD, line];

   -- Put text item on line 2
   line ← FormWindow.AppendLine [window];
   FormWindow.AppendItem[window, MyItems.text.ORD, line];
   };
```

29.3.3 Specifying Bitmaps in Choice Items

This example creates a choice item with three possible values. Two of them are bitmaps, one is a string. The initial value to be highlighted is #2, the string.

```
--The bits. (These are in a global frame or a file. They MUST be around for the duration of
the FormWindow since the bits are NOT copied.)
bm1:FormWindow.Bitmap ←[height:48, width:64, bitsPerLine:64, bits:[@bitmap1[0],0, 0]];
bm2:FormWindow.Bitmap ←[height:48, width:64, bitsPerLine:64, bits:[@bitmap2[0],0, 0]];
bitmap1: ARRAY [0..192) OF WORD ← [--some bits--];
bitmap2: ARRAY [0..192) OF WORD ← [--some bits--];
choiceOther: XString.ReaderBody ← XString.FromSTRING["OTHER"];
choices: ARRAY [0..3) OF FormWindow.ChoiceItem ← [
   [bitmap[0, bm1] ],
   [bitmap[1, bm2] ],
   [string[2, choiceOther] ]
   ];

FormWindow.MakeChoiceItem[
   window: window,
   tag: @tag,
   myKey: MyItems.choice.ORD,
   values: DESCRIPTOR[choices],
changeProc: ChoiceChangeProc,
initChoice: 2];
```

29.3.4 The NEXT Key and Text Items

This example creates a text item that inserts a new item after itself every time the user presses the NEXT key.

```
--Make the text item
MakeItems: FormWindow.MakeItemsProc =
  BEGIN
  . . .

  FormWindow.MakeTextItem[
    window: window,
    myKey: MyItems.text.ORD,
    width: 50,
    tag: @tag,
    initString: @initStringLong,
    nextOutOfProc: TextNextOut];
  . . .

  END;

TextNextOut: FormWindow.NextOutOfProc =
  BEGIN
  tag: XString.ReaderBody ← XString.FromSTRING["Inserted Item:"L];
  initString: XString.ReaderBody ← XString.FromSTRING["I DARE you! Edit ME!"L];

  --create a new line on which to display the new item
  nextLine: FormWindow.Line ← FormWindow.LayoutInfoFromItem
    [window, MyItems.testChoice2.ORD].line;
  line: FormWindow.Line ← FormWindow.InsertLine[window, nextLine, 60];

  --create the new item.
  FormWindow.MakeTextItem[
    window: window,
    myKey: cntr,
      --cntr is a counter to keep track of the next available
      --   key number since all ItemKeys are unique
    width: 50,
    tag: @tag,
    initString: @initString ,
    nextOutOfProc: TextNextOut];

  --put the new item on the line
  FormWindow.AppendItem[
    window: window,
    item: cntr,
    line: line];
  cntr ← cntr + 1;
```

```
--set the last item's NextOutOfProc to NIL
[] ← FormWindow.SetNextOutOfProc[window, item, NIL];
END;
```

29.3.5 Window Items (Including Interaction with the NEXT Key)

This example creates a window item that wishes to be given control when a user NEXTs into it.

```
--create the item
MakeItems: FormWindow.MakeItemsProc =
  BEGIN
  dims: Window.Dims ← [200,200];
  ...
  myWindow ← FormWindow.MakeWindowItem[
    window: window,
    myKey: MyItems.window.ORD,
    tag: @tag,
    size: dims,
    destroyProc: NIL,
    nextIntoProc: MyNextInto];

  --set the display and notify procs
  [] ← Window.SetDisplayProc[myWindow, WindowItemDisplayProc];
  [] ← TIP.SetTableAndNotifyProc [window: myWindow,
    table: TIPStar.NormalTable[], notify: MyNotify];
  . . .
  END;

--MyNextInto is called when a user presses the NEXT key "into" the window item
MyNextInto: FormWindow.NextIntoProc =
  BEGIN
  --set the input focus so the window item gets all of the notifications
  TIP.SetInputFocus [w: myWindow, takesInput: TRUE];
  END;

--FormWindow is notified so the window item no longer requires the NEXT key so
FormWindow can pass it along to the appropriate item

MyNotify: TIP.NotifyProc =
  BEGIN
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      atom = > SELECT z.a FROM
         nextDown = >
            FormWindow.TakeNEXTKey
               [window: myWindow.GetParent, item: MyItems.window.ORD];
            . . .
         ENDCASE;
      ENDCASE;
      ENDLOOP;
   END;
```

29.3.6 Hints

This example creates a text item that has a popup menu associated with it:

```
MakeItems: FormWindow.MakeItemsProc =
   BEGIN
   . . .
   FormWindow.MakeTextItem[
      window: window,
      myKey: MyItems.text.ORD,
      width: 50,
      tag: @tag,
      initString: @initString,
      hintsProc: TextHints];
   . . .
   END;
```

--Every time TextHints is called, specify the strings to put into the popup menu. The hintAction specifies that when a string is selected from the hints menu, it should replace the string in the text item

```
TextHints: FormWindow.TextHintsProc =
   BEGIN
   hintsArray ← --some computation--;
   RETURN [hints: DESCRIPTOR[hintsArray], freeHints: FreeHints, hintAction: replace];
   END;

FreeHints: FormWindow.FreeTextHintsProc =
   BEGIN
   --free the strings and whatever other storage here
   END;
```

29.3.7 Saving and Restoring Items

The following example saves the original values of the items in a form window and restores them when the user presses RESET.

```
--When creating the FormWindow also call
FormWindow.Save[window];

--user changes some values
--user decides he wants the original values back; presses Reset
FormWindow.Restore[window];
```

29.4Index of Interface Items

30 FormWindowMessageParse

30.1 Overview

The **FormWindowMessageParse** interface provides procedures that parse strings to produce various **FormWindow** TYPES. These strings are usually acquired from a message file. Currently, only **FormWindow.ChoiceItems** are supported.

30.2 Interface Items

```
ParseChoiceItemMessage: PROCEDURE [
   choiceItemMessage: XString.Reader,
   zone: UNCOUNTED ZONE]
RETURNS [choiceItems:FormWindow.ChoiceItems];
```

Parses a **choiceItemMessage** (presumably retrieved using **XMessage.Get**) with the following syntax: "choiceString:choiceNumber@choiceString:choiceNumber@|", where choiceString is the string to be displayed for that choice, choiceNumber is the fixed number associated with that choice, @ is the separator between choices, and | indicates the point at which to wrap the choices. The choices are displayed in the order they appear in the message. **choiceItems** is a descriptor for an array that must be freed by using **FreeChoiceItems**.

```
FreeChoiceItems: PROCEDURE [
   choiceItems:FormWindow.ChoiceItems,
   zone: UNCOUNTED ZONE];
```

Frees the array and everything it points to (strings).

30.3 Usage/Examples

The following example is taken from the folder implementation. The message acquired by **XMessage.Get** looks like "Sorted:0@Unsorted:1".

```
choices: FormWindow.ChoiceItems ← FormWindowMessageParse.ParseChoiceItemMessage [
   XMessage.Get[mh, FolderOps.kpsSorted], z];
```

```
FormWindow.MakeChoiceItem [
 window: window,
 myKey: MyItemType.sorted.ORD,
 values: choices,
 initChoice: sorted.ORD,
 changeProc: SortedChanged ];

FormWindowMessageParse.FreeChoiceItems[choices, z];
```

30.4 Index of Interface Items

31

IdleControl

31.1 Overview

The **IdleControl** interface provides access to ViewPoint's basic controlling module.

ViewPoint's control loop is organized as a series of two out-calls to a greeter procedure and a desktop procedure. Each procedure is implemented as a procedure variable, initialized to an appropriate no-op.

Interface procedures allow the client to plug in its own greeter and desktop procedures. A plugged-in procedure is then called the next time that the control routine goes around the loop.

31.2 Interface Items

IdleControl keeps track of one **GreeterProc** and a list of **DesktopProcs**. A client may plug in a number of **DesktopProcs** and specify the one to be called by the value of the **Atom.ATOM** returned by the **GreeterProc**.

31.2.1 DesktopPlug-in

DesktopProc: TYPE = PROCEDURE;

SetDesktopProc: PROCEDURE [atom: Atom.ATOM, desktop: DesktopProc] RETURNS [old: DesktopProc];

SetDesktopProc allows the client to specify the desktop procedure to be called in the control loop. **desktop** is the procedure to be called. **atom** is the **Atom.ATOM** associated with **desktop**. **old** is the previously plugged-in desktop procedure.

GetDesktopProc: PROCEDURE [atom: Atom.ATOM] RETURNS [DesktopProc];

31.2.2 Greeter Plug-in

GreeterProc: TYPE = PROCEDURE RETURNS [Atom.ATOM];

```
SetGreeterProc: PROCEDURE [new: GreeterProc] RETURNS [old: GreeterProc];
```

SetGreeterProc allows the client to specify the greeter procedure to be called in the control loop. **new** is the procedure to be called. **old** is the previously plugged-in greeter procedure.

```
GetGreeterProc: PROCEDURE RETURNS [GreeterProc];
```

```
DoTheGreeterProc: GreeterProc;
```

DoTheGreeterProc calls the currently plugged-in **GreeterProc**.

31.2.3 Idle Loop

The control loop is the logical equivalent of:

```
DO
  atom: Atom.ATOM ← pluggedInGreeterProc [];
  pluggedInDesktopProc ← GetDesktopProcWithAtom[atom];
  pluggedinDesktopProc[];
  ENDLOOP;
```

```
Idle: PROCEDURE
```

Idle is called or FORKed to enter the idle state. Only clients who start the world should call **Idle**.

31.3 Usage/Examples

In the following example, the **GreeterProc** (**IdleProc**) displays a bouncing square on the screen. The **GreeterProc** is set in the mainline code of the module. The **DesktopProc** and **GreeterProc** can be initialized in different modules as long as they agree on the **Atom.ATOM** (in this case **StarDesktop**).

```
starDesktopAtom: Atom.ATOM ← Atom.MakeAtom["StarDesktop"L];

IdleProc: IdleControl.GreeterProc = BEGIN --Display a bouncing square until the user
presses any key
   RETURN [starDesktopAtom];
   END;

StarDesktop: PROCEDURE = BEGIN
--Do Star logon
--Initialize and display Star desktop
--Wait until Logoff
 END;

Init: PROCEDURE =
   BEGIN
   [] ←IdleControl.SetGreeterProc[IdleProc];
   [] ←IdleControl.SetDesktopProc [starDesktopAtom, StarDesktop];
   END; -- of Init
```

31.4 Index of Interface Items

32

KeyboardKey

32.1 Overview

KeyboardKey is a keyboard registration facility. It provides clients with a means of registering "system-wide" keyboards (available all the time, like English, French, European), a special keyboard (like Equations), and/or client-specific keyboards (such as these available only when the client has the input focus). The labels from these registered keyboards are displayed in the soft keys when the user holds down the KEYBOARD key.

The client adds system keyboards by calling **AddToSystemKeyboards**. The client registers a special keyboard by calling **RegisterClientKeyboards** with the **SPECIALKeyboard** parameter. The client registers client-specific keyboards by calling **RegisterClientKeyboards** with the **keyboards** parameter.

32.2 Interface Items

32.2.1 System Keyboards

A *system keyboard* is a one that is available to all clients who wish to recognize some general set of keyboards. (The default case is for a client to recognize system keyboards.) Examples of system keyboards are the various language keyboards--English, French, European,and so forth, and the general-purpose keyboards--Math, Office, Logic, and Dvorak.

AddToSystemKeyboards: PROCEDURE [**keyboard**: BlackKeys.**Keyboard**];

The **AddToSystemKeyboards** procedure registers a client's keyboard interpretation with the keyboard key manager. The client is expected to provide a pointer to a keyboard record. This keyboard is made available whenever system keyboards are available.

May raise **Error[alreadyInSystemKeyboards]**.

RemoveFromSystemKeyboards: PROCEDURE [**keyboard**: BlackKeys.**Keyboard**];

Removes a **Keyboard** from the list of system keyboards.

May raise **Error[notInSystemKeyboards]**.

32.2.2 Client Keyboards

A *client keyboard* is one that is specific to the application and has no meaning in a different context. Examples are the special keyboards (such as equations and fields) and Spreadsheet and 3270 keyboards.

A client registers its keyboards with the keyboard manager when it gets control (gets the input focus). **RegisterClientKeyboards** tells the keyboard manager what keyboards should be made available to the user when the KEYBOARD key is held down. When the client loses control (releases the input focus) it should call **RemoveClientKeyboards** to release its keyboards. Only 0-1 set of client keyboards is registered at any given time. If no client is registered, then all system keyboards are available to the user.

```
RegisterClientKeyboards: PROCEDURE [
  wantSystemKeyboards: BOOLEAN ← TRUE,
  SPECIALKeyboard: BlackKeys.Keyboard ← NIL.
  keyboards: LONG DESCRIPTOR FOR ARRAY OF BlackKeys.KeyboardObject ← NIL];
```

RegisterClientKeyboards establishes a list of client keyboards with the keyboard manager. This should occur at the same time the client takes the input focus. **wantSystemKeyboards** specifies whether the client wishes to recognize system keyboards. **SPECIALKeyboard** denotes the keyboard to be invoked by pressing the key combination of KEYBOARD key and the soft key labeled "Special". The **keyboards** array contains any other client keyboards. A client typically provides only a Special keyboard and **wantSystemKeyboards** = TRUE. If **wantSystemKeyboards** = FALSE ,the client should set one of his keyboards using **SetKeyboard** (see section 32.2.3).

```
RemoveClientKeyboards: PROCEDURE ;
```

RemoveClientKeyboards removes the client's keyboards from the keyboard manager's list. This list of available keyboards reverts to system keyboards only. The "Set " keyboard is the last system keyboard that was set (either by the user or a call to **SetKeyboard**). It is the client's responsibility to make sure his keyboards are removed when relinquishing control. It is appropriate for this to be done as part of a **TIP.LosingFocusProc**.

32.2.3 Setting and Enumerating Keyboards

Note: Most clients will probably not need to use the information in this section.

```
SetKeyboard: PROCEDURE [keyboard: BlackKeys.Keyboard];
```

SetKeyboard sets the current keyboard to **keyboard**. This keyboard remains the current keyboard until the user presses a KEYBOARD key/SoftKeyOption/Set combination, which chooses a new keyboard, or until another **SetKeyboard** command is encountered.

SetKeyboard is provided for those clients who have reason to set a keyboard programmatically. The usual case is for the user to set a keyboard by pressing the key combination KEYBOARD key/SomeSoftKeyDesignatingAKeyboard. However, for a client calling **RegisterClientKeyboards** with **wantSystemKeyboards** = FALSE it is appropriate to call **SetKeyboard[@oneOfClientKeyboards]** . (Otherwise the user could not type until he made a keyboard choice through the KEYBOARD key/SoftKey routine.) The other primary reason for calling **SetKeyboard** is to set an initial keyboard at boot time.

```
EnumerateKeyboards: PROCEDURE [class: KeyboardClass, enumProc: EnumerateProc];

EnumerateProc: TYPE = PROCEDURE[keyboard: BlackKeys.Keyboard, class: KeyboardClass]
  RETURNS[stop: BOOLEAN ← FALSE];
```

EnumerateKeyboards calls the specified **EnumerateProc** until the **Stop** boolean becomes TRUE or until there are no more **keyboards** to enumerate. When calling **EnumerateKeyboards,** the client may specify which keyboards he wants enumerated: **system**, **client**, **special**, or **all** of the registered keyboards. When the keyboard manager calls the client's **EnumerateProc**, the keyboard returned is qualified by **class**: **client**, **system**, or the **special** keyboard..

```
KeyboardClass: TYPE = {system, client, special, all, none};
```

system =	A system keyboard is one that is available to all clients who wish to recognize some general set of keyboards. Examples of system keyboards are the various language keyboards - English, French, European, etc., and the general-purpose keyboards--Math, Office, Logic, and Dvorak.
client =	A client keyboard is one that is specific to the application. These are the keyboards registered in the **keyboards** array by the client calling **RegisterClientKeyboards**.
special =	A client-specific keyboard is invoked by pressing the combination of KEYBOARD key and the soft key labeled "Special". Specifically, this is the keyboard registered by the client as **SPECIALKeyboard** when calling **RegisterClientKeyboards**.
all =	All keyboards: system, client, and special.

32.2.4 Alternate Keyboard

```
SetFirstAltKb: PUBLIC PROC [
  class: KeyboardKey.KeyboardClass ← client, r: XString.Reader ← NIL];
```

Allows clients to establish a first alternate keyboard selection that will be installed when the user presses the keyboard key. **r** is the name of the keyboard

There are two alternate keyboard possibilities:

(1) **system** => The BasicWorkstation scans the User Profile at each logon for a [System] First Alternate Keyboard entry. The system first alternate keyboard will be set based on this entry (none if no entry is found).
Client code can also set the system first alternate keyboard by calling **SetFirstAltKb**[system, reader]. Though this should be done thoughtfully in a way that does not interfere with the user or the systems intentions.

Since the special keyboard is a special kind of client keyboard that is known to the system, calling **SetFirstAltKb**[special] will set the system first alternate keyboard to the special keyboard.

(2).**client** => The client may set a client first alternate keyboard by calling **SetFirstAltKb**[client, reader]. Note in the paragraph above that the special keyboard can be set as a system first alternate as well as a client first alternate.

When the user presses the keyboard key if there is a client first alternate keyboard it will be installed else if there is a system first alternate keyboard it will be installed.

When the user releases the keyboard key (assuming no action was taken to change or set a keyboard) any alternate keyboard that was installed will be removed leaving the keyboard in the same state as it was before the keyboard key was pressed.

```
GetFirstAltKb: PUBLIC PROC [
   class: KeyboardKey.KeyboardClass ← client]
   RETURNS [r: XString.Reader ← NIL];
```

Returns the name of the first alternate keyboard of **class**. Any client wishing to temporarily set the system first alternate keyboard should first get the current one so that it may be reset at the appropriate time.

32.2.5 Keyboard Window Plug-in

This section pertains only to those clients interested in implementing a keyboard window facility.

```
ShowKeyboardProc: TYPE = PROCEDURE;

SetShowKeyboardProc: PROCEDURE [ShowKeyboardProc];

GetShowKeyboardProc: PROCEDURE RETURNS [ShowKeyboardProc];
```

SetShowKeyboardProc and **GetShowKeyboardProc** provide an interface between a keyboard window application and **KeyboardKey**'s "Show" key. The clients **ShowKeyboardProc** is called whenever the user presses the key combination KEYBOARD key/Show. This gives the client the opportunity to display a keyboard window.

32.2.6 Errors

```
Error: ERROR[code: ErrorCode];

ErrorCode: TYPE = {alreadyInSystemKeyboards, notInSystemKeyboards,
   insufficientSpace};
```

32.3 Usage/Examples

32.3.1 AddToSystemKeyboards Example

In this application, a keyboard that will be useful across all applications has been defined. Registering it as a system keyboard will make it available globally.

```
usefulKeyboard: BlackKeys.KeyboardObject ←
  [charTranslator: [proc: MyCharTrans, data: NIL],
   pictureProc: MapPicture,
  label ← XString.FromString["Useful Keyboard"L]];

KeyboardKey.AddToSystemKeyboards[@myUsefulKeyboard];
```

The keyboard manager adds the keyboard **usefulKeyboard** to the list of registered system keyboards and a key labeled *Useful Keyboard* to its labels for the **KeyboardKey** soft keys. When the user pushes the soft key labeled *Useful Keyboard*, **MyCharTrans** will be registered as the **TIP.CharTranslator**, and if the keyboard window is open, **MapPicture** is called so that the picture and geometry table can be mapped.

32.3.2 Special Keyboard Example

This example contains a keyboard that is specific to a particular application and is available to the user through the Special key. The user should also be able to use the system keyboards in this application. Notice that it is appropriate to default the **label** when specifying a Special keyboard, because this keyboard is presented to the user as the current Special keyboard and is labeled as such.

```
AddMyClientKeyboards: PROCEDURE =
BEGIN
 specialKeyboard: BlackKeys.KeyboardObject;
 fileName: XString.ReaderBody ← XString.FromSTRING["JSpecial.TIP"L];
 table: TIP.Table ← TIP.CreateTable[@fileName];
 [] ← TIP.SetNotifyProcForTable[table, NotifyProc];
 specialKeyboard ← [table: table];
 KeyboardKey.RegisterClientKeyboards[
  wantSystemKeyboards: TRUE,
   SPECIALKeyboard: @specialKeyboard];
END; -- AddMyClientKeyboards

 LosingFocusProc: TIP.LosingFocusProc =
 <<[w: Window.Handle, data: LONG POINTER]>>
 BEGIN
  KeyboardKey.RemoveClientKeyboards[];
  --release any data structures I don't want to keep around
 END; -- LosingFocusProc
```

32.3.3 Registering Multiple Client Keyboards Example

This example deals with a client who has a special keyboard and several client-specific keyboards and does not plan to allow the user to use any system keyboards while in this application.

```
keyboardRecords: ARRAY [0..3) OF BlackKeys.KeyboardObject;        -- Records filled in
specialKeyboard: BlackKeys.Keyboard;                               -- elsewhere

AddClientKeyboards: PROCEDURE =
BEGIN
 KeyboardKey.RegisterClientKeyboards[
   wantSystemKeyboards: FALSE,
   SPECIALKeyboard: specialKeyboard,
   keyboards: DESCRIPTOR[keyboardRecords]];
 KeyboardKey.SetKeyboard[@keyboardRecords[0]]
END; -- AddClientKeyboards

LosingFocusProc: TIP.LosingFocusProc =
 <<[w: Window.Handle, data: LONG POINTER]>>
 BEGIN
  KeyboardKey.RemoveClientKeyboards[];
  --release any data structures I don't want to keep around
 END; -- LosingFocusPro --
```

32.4 Index of Interface Items

33

KeyboardWindow

33.1 Overview

The **BlackKeys** and **KeyboardKey** interfaces provide the framework for including a keyboard window in ViewPoint. The window implementation is a plug-in (see KeyboardKey.SetShowKeyboardProc). This **KeyboardWindow** interface and its implementation provide one such keyboard window. This **KeyboardWindow** interface also provides a number lock key state.

33.2 Interface Items

33.2.1 Default Values

defaultPicture: BlackKeys.Picture;

defaultGeometry: BlackKeys.GeometryTable;

The default values provided by this keyboard window implementation correspond to the standard English keyboard.

DefaultPictureProc: BlackKeys.PictureProc;

DefaultPictureProc returns **defaultPicture** and **defaultGeometry** to the caller when **action = acquire**. Clients may specify **pictureProc: KeyboardWindow.DefaultPictureProc** in their **BlackKeys.KeyboardObject** if they wish to display the default picture in the keyboard window while their keyboard is in effect.

picture = BlackKeys.nullPicture or **BlackKeys.PictureProc = NIL** will result in the keyboard window displaying only gray in the viewing region.

33.2.2 Geometry Table Structure

```
GeometryTableEntry: TYPE = RECORD[
 box: Box, key: KeyStations, shift: ShiftState];

Box: TYPE = RECORD[place: Window.Place, width: INTEGER, height: INTEGER];
```

Area within the bitmap that generates a particular keystroke when selected with the mouse.

```
KeyStations: TYPE = MACHINE DEPENDENT {
 k1, k2, k3, k4, k5, k6, k7, k8, k9, k10, k11, k12, k13, k14, k15, k16, k17,
 k18, k19, k20, k21, k22, k23, k24, k25, k26, k27, k28, k29, k30, k31, k32,
 k33, k34, k35, k36, k37, k38, k39, k40, k41, k42, k43, k44, k45, k46, k47,
 k48, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, last(96)};
```

The following is a translation to LevelIVKeys.KeyName:

```
k1 => One;
k2 => Q;
k3 => A;
k4 => Two;
k5 => Z;
k6 => W;
k7 => S;
k8 => Three;
k9 => X;
k10 => E;
k11 => D;
k12 => Four;
k13 => C;
k14 => R;
k15 => F;
k16 => Five;
k17 => V;
k18 => T;
k19 => G;
k20 => Six;
k21 => B;
k22 => Y;
k23 => H;
k24 => Seven;
k25 => N;
k26 => U;
k27 => J;
k28 => Eight;
k29 => M;
k30 => I;
k31 => K;
k32 => Nine;
k33 => Comma;
k34 => O;
k35 => L;
```

```
k36 => Zero;
k37 => Period;
k38 => P;
k39 => SemiColon;
k40 => Minus;
k41 => Slash;
k42 => LeftBracket;
k43 => CloseQuote;
k44 => Equal;
k45 => RightBracket;
k46 => OpenQuote;
k47 => Key47;
k48 => Tab;
a1 => ParaTab;
a2 => BS;
a3 => Lock;
a4 => NewPara;
a5 => LeftShift;
a6 => RightShift;
a7 => Space;
a8 => A8;
a9 => A9;
a10 => A10;
a11 => A11;
a12 => A12;
```

```
ShiftState: TYPE = {None, One, Two, Both};
```

Simulates the position of the shift keys associated with the keystroke.

33.2.3 Bitmap Structure

BlackKeys.Picture.bitmap is a LONG POINTER. It is further defined within this keyboard window implementation as follows: **bitmap** points to the bits of the keyboard bitmap where dims = [505, 145] and **bitmapBitWidth** = 32*16.

33.2.4 Getting to the Keyboard Window Handle

```
GetDisplayWindow: PROCEDURE RETURNS [Window.Handle];
```

Returns handle to the body window of the keyboard window.

33.2.5 The Number Lock Key State

```
NumLockState: TYPE = {set, reset};

numLockState: NumLockState;

SetNumLockState: PROCEDURE [newNumLockState: NumLockState];
```

The number lock key is typically found on the ten key pad. **NumLockState** indicates the states the number lock key can exist. **set** means the ten key pad is in number mode. **reset** means the ten key pad is in cursor key mode. **numLockState** is the current state of the number lock key. Clients who wish to change the value of numLockState can utilize **SetNumLockState**.

33.3 Usage/Examples

33.3.1 Using DefaultPictureProc

```
DefineKeyboard: PROCEDURE =
BEGIN
 nameString: XString.ReaderBody ← XString.FromSTRING["Zulu"L]

 zuluKeyboardRecord: BlackKeys.KeyboardObject ← [
  table: NIL,
  charTranslator: [MakeChar, NIL],
  pictureProc: KeyboardWindow.DefaultPictureProc,
  label: XString.CopyToNewReaderBody[@nameString, Heap.systemZone]];
 --Save the pointer to the record somewhere for future use --
END; --DefineKeyboard --
```

33.3.2 Using defaultGeometry

```
DefineKeyboard: PROCEDURE =
BEGIN
 nameString: XString.ReaderBody ← XString.FromSTRING["Swahili"L]

 swahiliKeyboardRecord: BlackKeys.KeyboardObject ← [
  table: NIL,
  charTranslator: [MakeChar, NIL],
  pictureProc: MapBitmapFile,
  label: XString.CopyToNewReaderBody[@nameString, Heap.systemZone]];
 --Save the pointer to the record somewhere for future use --
END; --DefineKeyboard --

MapBitmapFile: BlackKeys.PictureProc =
BEGIN
 pixPtr: BlackKeys.Picture.bitmap ← BlackKeys.nullPicture;
 SELECT action FROM
  acquire = >
{--Do the right thing to map the bitmap. Uses the default Geometry table --.
   RETURN[pixPtr, KeyboardWindow.defaultGeometry] };
  release = > {--Do the right thing to unmap the bitmap
   RETURN[BlackKeys.nullPicture, NIL] }
END; -- MapBitmapFile
```

33.3.3 Sample Geometry Table Entries

```
box: [place: [x: XXX, y: XXX], w idth: XXX, height: XXX], key: XXX, shift: XXX
[[19, 11], 27, 27], k48, None           -- 'tab' key, dims: whole key picture
[[51, 11], 27, 14], k1, One             -- shifted '1' key, dims: upper half key
[[51, 24], 27, 14], k1, None            -- '1' key, dims: lower half key
[[83, 11], 27, 14], k4, One             -- shifted '2' key, dims: upper half key
[[83, 24], 27, 14], k4, None            -- '2' key, dims: lower half key
```

33.4 Index of Interface Items

34

LevelIVKeys

34.1 Overview

LevelIVKeys is documented in the *Pilot Programmer's Manual* (610E00160); however, the names of several keys were changed for ViewPoint. The key names now more closely match the names on the physical keys.

34.2 Interface Items

```
OPEN ks: KeyboardWindow.KeyStations;

DownUp: TYPE = ks.DownUp;

Bit: TYPE = ks.Bit;

KeyBits: TYPE = PACKED ARRAY KeyName OF DownUp;

KeyName: TYPE = MACHINE DEPENDENT {
  notAKey(0),
  Keyset1(ks.KS1), Keyset2(ks.KS2), Keyset3(ks.KS3), Keyset4(ks.KS4),
  Keyset5(ks.KS5),
  MouseLeft(ks.M1), MouseRight(ks.M3), MouseMiddle(ks.M2),
  Five(ks.k16), Four(ks.k12), Six(ks.k20), E(ks.k10), Seven(ks.k24),
  D(ks.k11), U(ks.k26), V(ks.k17), Zero(ks.k36), K(ks.k31), Minus(ks.k40),
  P(ks.k38), Slash(ks.k41), Font(ks.R8), Same(ks.L8), BS(ks.A2),
  Three(ks.k8), Two(ks.k4), W(ks.k6), Q(ks.k2), S(ks.k7), A(ks.k3),
  Nine(ks.k32), I(ks.k30), X(ks.k9), O(ks.k34), L(ks.k35), Comma(ks.k33),
  CloseQuote(ks.k43), RightBracket(ks.k45), Open(ks.L11), Keyboard(ks.R11),
  One(ks.k1), Tab(ks.k48), ParaTab(ks.A1), F(ks.k15), Props(ks.L12),
  C(ks.k13), J(ks.k27), B(ks.k21), Z(ks.k5), LeftShift(ks.A5),
  Period(ks.k37), SemiColon(ks.k39), NewPara(ks.A4),
  OpenQuote(ks.k46), Delete(ks.L3), Next(ks.R1), R(ks.k14), T(ks.k18),
  G(ks.k19), Y(ks.k22), H(ks.k23), Eight(ks.k28), N(ks.k25), M(ks.k29),
  Lock(ks.A3), Space(ks.A7), LeftBracket(ks.k42), Equal(ks.k44),
RightShift(ks.A6), Stop(ks.R12), Move(ks.L9), Undo(ks.R6), Margins(ks.R5),
R9(ks.R9), L10(ks.L10), L7(ks.L7), L4(ks.L4), L1(ks.L1), A9(ks.A9),
R10(ks.R10), A8(ks.A8), Copy(ks.L6), Find(ks.L5), Again(ks.L2),
Help(ks.R2), Expand(ks.R7), R4(ks.R4), D2(ks.D2), D1(ks.D1),
Center(ks.T2), T1(ks.T1), Bold(ks.T3), Italics(ks.T4), Underline(ks.T5),
Superscript(ks.T6), Subscript(ks.T7), Smaller(ks.T8), T10(ks.T10),
R3(ks.R3), Key47(ks.k47), A10(ks.A10), Defaults(ks.T9), A11(ks.A11),
A12(ks.A12)};
```

34.3 Index of Interface Items

35

MenuData

35.1 Overview

The **MenuData** interface defines the data abstraction that is a titled list of named commands. It defines the object formats for a menu item and a menu as well as how to create and manipulate these objects. It is not concerned with how a menu might be displayed to a user.

35.2 Interface Items

35.2.1 Menu and Item Creation

Items and menus are the two primary data objects in the **MenuData** interface. *Items* are a name-procedure pair that constitute a command. *Menus* are an abstraction representing a collection of items to be presented to the user. These objects can be built and deallocated through this interface.

```
CreateItem: PROCEDURE [
    zone: UNCOUNTED ZONE,
    name:XString.Reader,
    proc: MenuProc,
    itemData: LONG UNSPECIFIED ← 0]
    RETURNS [ItemHandle];

ItemHandle: TYPE = LONG POINTER TO Item;

Item: TYPE = PrivateItem;

MenuProc: TYPE = PROCEDURE [
    window: Window.Handle, menu: MenuHandle, itemData: LONG UNSPECIFIED];
```

CreateItem builds an item record in the indicated **zone** to be added to a menu. The **name** parameter is copied so it can be in the client's local frame. The **proc** parameter is the command procedure that is associated with a command name in an item. Client data that must be available when the **MenuProc** is called can be passed via the **itemData** parameter. **zone** can be NIL, in which case **MenuData** supplies its own zone (see **PublicZone** below).

An **Item** is the representation for a {command-name, command-procedure} pair. The "nameWidth" field, if non-zero, is the display width of the name. It may be set by a module that computes the width using **SetItemNameWidth** (see §35.2.3). Except for that, an item is read-only.

```
DestroyItem: PROCEDURE [zone: UNCOUNTED ZONE, item: ItemHandle];
```

This procedure destroys the item, recovering the space. **zone** must be the zone in which the item was created, and **item** is the **ItemHandle** returned by **CreateItem**. The item should not be in use when this procedure is called.

```
CreateMenu: PROCEDURE [
    zone: UNCOUNTED ZONE,
    title: ItemHandle,
    array: ArrayHandle,
    copyItemsIntoMenusZone: BOOLEAN ← FALSE ]
    RETURNS [MenuHandle];

ArrayHandle: TYPE = LONG DESCRIPTOR FOR ARRAY OF ItemHandle;

MenuHandle: TYPE = LONG POINTER TO MenuObject;

MenuObject: TYPE = PrivateMenu;
```

CreateMenu builds a menu record in **zone**. **title** is an item containing the menu's title. **array** contains the collection of items that make up the menu. The items pointed to by the **array** and the **title** parameters are copied only if **copyItemsIntoMenusZone** is TRUE. Because item records are read-only, an item can be in several menus without copying. The procedure associated with the **title** item is currently unused and should be NIL for future compatibility. **zone** can be NIL, in which case **MenuData** supplies its own zone (see **PublicZone** below).

```
DestroyMenu: PROCEDURE [zone: UNCOUNTED ZONE, menu: MenuHandle ]
```

DestroyMenu destroys the menu, recovering the space. **zone** must be the zone in which the menu was created; **menu** is the **MenuHandle** returned by **CreateMenu**. It should only be called when the menu is not in use. There is no explicit way to test if a menu is in use.

```
PublicZone: PROCEDURE RETURNS [UNCOUNTED ZONE];
```

PublicZone returns the zone used by **MenuData** when **CreateItem** or **CreateMenu** is called with **zone** = NIL.

35.2.2 Menu Manipulation

```
AddItem: PROCEDURE [menu: MenuHandle, new: ItemHandle ] =
    INLINE {menu.swapItemProc [menu: menu, old: NIL, new: new]};
```

```
SubtractItem: PROCEDURE [ menu: MenuHandle, old: ItemHandle ] =
    INLINE {menu.swapItemProc [menu: menu, old: old, new: NIL]};
```

```
SwapItem: PROCEDURE [ menu: MenuHandle, old, new: ItemHandle ] =
    INLINE {menu.swapItemProc [menu: menu, old: old, new: new]};
```

These procedures alter a menu in the obvious ways. They call through the **swapItemProc** field in the menu object. This allows a module that posts a menu to "plant" a procedure in the **swapItemProc** field and thus get control on add/subtract/swap requests. With control, data can be monitored appropriately.

```
SetSwapItemProc: PROCEDURE [menu: MenuHandle, new: SwapItemProc]
    RETURNS [old: SwapItemProc];
```

```
SwapItemProc: TYPE = PROCEDURE [menu: MenuHandle, old, new: ItemHandle];
```

The **SwapItemProc** is the work horse for manipulating menus, as evidenced by the INLINE calls above. It can be changed by calling **SetSwapItemProc**.

```
UnpostedSwapItemProc: SwapItemProc;
```

This procedure is the standard one that is placed in a menu's **swapItemProc** when the menu is created. It is in the **MenuData** implementation, and it can handle altering a menu when it is not posted. As discussed above, if a routine that posts a menu wants to get control of attempted menu alterations, it should alter the **swapItemProc** in the menu. When it has finished posting the menu, it should store **MenuData.UnpostedSwapItemProc** as the **swapItemProc**. Alternatively, it can call **MenuData.UnpostedSwapItemProc** from within its own **swapItemProc** to perform the actual alteration of the menu object.

35.2.3 Accessing Data

The following provide procedural access to the internal data structures for an item or menu.

```
ItemData: PROCEDURE [item: ItemHandle] RETURNS [LONG UNSPECIFIED];
```

```
ItemName: PROCEDURE [item: ItemHandle]
    RETURNS [name: XString.ReaderBody];
```

```
SetItemNameWidth: PROCEDURE [item: ItemHandle, width: CARDINAL] =
    INLINE {item.nameWidth ← width};
```

```
ItemNameWidth: PROCEDURE [item: ItemHandle] RETURNS [CARDINAL] =
    INLINE {RETURN [item.nameWidth]};
```

```
ItemProc: PROCEDURE [item: ItemHandle] RETURNS [proc: MenuProc] =
    INLINE {RETURN [item.proc]};
```

```
MenuArray: PROCEDURE [menu: MenuHandle] RETURNS [array: ArrayHandle] =
    INLINE {RETURN [menu.array]};
```

```
MenuTitle: PROCEDURE [menu: MenuHandle] RETURNS [title: ItemHandle] =
   INLINE {RETURN [menu.title]};
```

Note: MenuObjects and Items are private records that are of use to menu posters but not of interest to general clients. The private records shown below are purely informative.

```
PrivateItem: TYPE = PRIVATE RECORD [
   proc: MenuProc,
   nameWidth: NATURAL,
   nameBytes: NATURAL,
   body: SELECT hasItemData: BOOLEAN FROM
     FALSE = > [name XString.ByteSequence],
     TRUE = > [itemData: LONG UNSPECIFIED, name: XString.ByteSequence]
     ENDCASE];

PrivateMenu: TYPE = PRIVATE RECORD [
   zone: UNCOUNTED ZONE,
   swapItemProc: SwapItemProc,
   title: ItemHandle ←NIL,
   array: ArrayHandle ←NIL,
   arrayAllocatedItemHandles: NATURAL ←0,
   itemsInMenusZone: BOOLEAN ←FALSE];
```

35.3 Usage/Examples

A mechanism outside the scope of this interface displays a menu to the user. A given menu instance cannot ordinarily be displayed more than once at the same time.

When the user asks that a command be executed, the command item's procedure is called. The window argument is a pointer that is dependent on the display mechanism; it might be the StarWindowShell.Handle that the menu is posted in.

35.3.1 Example 1

```
sysZ: UNCOUNTED ZONE = Heap.systemZone;

Init: PROC = {
  sampleTool: XString.ReaderBody ←XString.FromSTRING["Sample Tool"L];
  Attention.AddMenuItem [
    MenuData.CreateItem [
      zone: sysZ,
      name: @sampleTool,
      proc: MenuProc] ] };

MenuProc: MenuData.MenuProc = {
  another: XString.ReaderBody ←XString.FromSTRING["Another"L];
  repaint: XString.ReaderBody ←XString.FromSTRING["Repaint"L];
  post: XString.ReaderBody ←XString.FromSTRING["Post A Message"L];
  sampleTool: XString.ReaderBody ←XString.FromSTRING["Sample Tool"L];

  -- Create the StarWindowShell. --
```

```
    shell: StarWindowShell.Handle = StarWindowShell.Create [name: @sampleTool];

    .
    .
    .

    -- Create some menu items. --
    z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
    items: ARRAY [0..3) OF MenuData.ItemHandle ← [
       MenuData.CreateItem [zone: z, name: @another, proc: MenuProc],
       MenuData.CreateItem [zone: z, name: @repaint, proc: RepaintMenuProc],
       MenuData.CreateItem [zone: z, name: @post, proc: Post] ];
    myMenu: MenuData.MenuHandle = MenuData.CreateMenu [
       zone: z,
       title: NIL,
       array: DESCRIPTOR [items] ];

    StarWindowShell.SetRegularCommands [sws: shell, commands: myMenu];
    .
    .
    .
    };

Post: MenuData.MenuProc = {
    msg: XString.ReaderBody ← XString.FromSTRING ["This is a sample attention window
    message."L];
    Attention.Post [@msg] };

RepaintMenuProc: MenuData.MenuProc = {
    body: Window.Handle = StarWindowShell.GetBody[[window]];
    Window.InvalidateBox[body, [[0, 0], [30000, 30000] ]];
    Window.Validate[body] };

-- Mainline code --

Init[];
```

35.3.2 Example 2

```
-- Declare and create an item title and command array to be placed in a menu --
mouseMenuTitle: MenuData.ItemHandle ← InitMouseMenuTitle [];
mouseMenuCmnds: ARRAY [0..10) OF MenuData.ItemHandle;

-- Create the menu --
mouseMenu: MenuData.MenuHandle ← MenuData.CreateMenu [
   zone: MenuData.PublicZone [],-- could be a client-supplied zone --
   title: mouseMenuTitle,
   array: DESCRIPTOR [@mouseMenuCmnds[0], 1] ];

CommandProc: MenuData.MenuProc = {
   --Does something reasonable for the corresponding item -- };
```

```
.
.
.

InitMouseMenuTitle: PROCEDURE RETURNS [MenuData.ItemHandle] = {
   zone: UNCOUNTED ZONE ← MenuData.PublicZone [];
   mouseBitMap: ARRAY [0..15) OF WORD ← [ -- ... octal code -- ];
   mouseSymbolChar: XString.Character ←
      SimpleTextFont.AddClientDefinedCharacter [ -- ... parameters -- ];
   mouseString: XString.ReaderBody ← XString.FromChar [@mouseSymbolChar];
   cmndTitle: XString.ReaderBody ← XString.FromSTRING ["Command"];
   mouseMenuCmnds[0] ← MenuData.CreateItem [zone, @cmndTitle, CommandProc];
   RETURN [MenuData.CreateItem [zone, @mouseString, NIL] ] };
```

The above example is just one technique for initializing a menu. The routine **InitMouseMenuTitle** places variables that don't need to be global in the local frame . Pay close attention to placement of variables to prevent dangling references.

35.4 Index of Interface Items

36 MessageWindow

36.1 Overview

MessageWindow provides a facility for posting messages in a window to the user. This is similar to posting messages using the **Attention** interface, but many message windows can be on the screen at once, while there is only one attention window. A message window is usually a short window with less than 10 lines of text in it. As more messages are posted, previous messages scroll off.

MessageWindow.Create takes a "plain" window, typically obtained by calling **StarWindowShell.CreateBody** or **FormWindow.MakeWindowItem**, and turns it into a message window. Messages may then be posted by calling **Post**. To clear the window, call **Clear**. Various TYPEs may be formatted into messages to be posted in the window by using the **XFormat.Object** returned by **XFormatObject**.

36.2 Interface Items

36.2.1 Create, Destroy, etc.

```
Create: PROCEDURE [window: Window.Handle,
   zone: UNCOUNTED ZONE ← NIL, lines: CARDINAL ← 10];
```

Create turns **window** into a message window. **zone** will be used for storage of any strings posted. If **zone** is NIL, a private zone is used. **lines** is the number of lines of text to display. After more than **lines** of text are posted, the oldest lines are scrolled out of the window. Fine point: The current ViewPoint implementation does not support user scrolling.

```
Destroy: PROCEDURE [Window.Handle];
```

Destroy turns the window back into an ordinary window, destroying any **MessageWindow** specific context associated with the window. It does not destroy the window.

```
IsIt: PROCEDURE [Window.Handle] RETURNS [yes: BOOLEAN];
```

IsIt returns TRUE if the window was made into a message window by a call to **Create**.

36.2.2 Posting messages

```
Post: PROCEDURE [window: Window.Handle,
  r: XString.Reader, clear: BOOLEAN ← TRUE];
```

Post displays **r** in **window**. If **clear** is TRUE, **r** starts on a new line. If **clear** is FALSE, **r** is appended to the last line posted.

```
PostSTRING: PROCEDURE [window: Window.Handle,
  s: LONG STRING, clear: BOOLEAN ← TRUE] = INLINE
    BEGIN
    r: XString.ReaderBody ← XString.FromSTRING [s];
    MessageWindow.Post [window, @r, clear];
    END;
```

PostSTRING posts **s** in **window**. If **clear** is TRUE, **r** starts on a new line. If **clear** is FALSE, **r** is appended to the last line posted.

```
Clear: PROCEDURE [window: Window.Handle];
```

Clear clears the entire **window**.

```
XFormatObject: PROCEDURE [window: Window.Handle] RETURNS [o: XFormat.Object];
```

XFormatObject returns an **XFormat.Object** that can be used to post messages in **window**. The format procedure logically calls **Post** with **clear** = FALSE. (See examples.)

36.3 Usage/Examples

In the following example, a client displays the name and size of a file. It uses the **NSFile** interface to access the file and get the name and size attributes. See the *Services Programmer's Guide* (610E00180)–*Filing Programmer's Manual* for documentation on the **NSFile** interface. The example intermixes use of the format handle and use of the **Post** procedure.

```
  ...
  msgW: Window.Handle ← FormWindow.MakeWindowItem [...];
  MessageWindow.Create [window: msgW, lines: 5];
  ...
  PostNameAndSize [file, msgW];
  ...

PostNameAndSize: PROCEDURE [file: NSFile.Handle, msgW: Window.Handle ] = {
  nameSelections: NSFile.Selections = [interpreted: [name: TRUE]];
  attributes: NSFile.AttributesRecord;
  msgWFormat: XFormat.Object ← MessageWindow.XFormatObject[msgW];
  rb: XString.ReaderBody ← Message[theFile];
  MessageWindow.Post[window: msgW, s: @rb, clear: TRUE]; -- start a new message
  XFormat.NSString[@msgWFormat, attributes.name];
  XFormat.ReaderBody[h: @msgWFormat, rb: Message[contains]];
  XFormat.Decimal[h: @msgWFormat, n: NSFile.GetSizeInBytes[file]];
rb ← Message[bytes];
MessageWindow.Post[window: msgW, s: @rb]}; -- clear defaults to TRUE

Message: PROCEDURE [key: {theFile, contains, bytes}] RETURNS [XString.ReaderBody] = {
   ...};
```

An example of the resulting message displayed in the message window is

```
The file Foo contains 53324 bytes
```

36.4 Index of Interface Items

37

OptionFile

37.1 Overview

OptionFile reads values from profile files (text files) with the following format:

[Section]
Entry1: TRUE -- a boolean entry
Entry2: A string value
Entry3: 123 -- an integer entry

These files are primarily used for keeping user options across logon and boot sessions (thus the name profile file). Applications typically read various options out of the current user profile file at logon. These options often specify default values for properties, the behavior of the application, or both.

37.2 Interface Items

37.2.1 Getting Values from a File

Each GetXXXValue procedure takes a section name and an entry name that identifies the entry. It is expected that the section and entry strings are obtained from XMessage. Each also takes a file. If file is defaulted, the current user profile is used (see the Current Profiles section below). All these procedures may raise Error [invalidParameters, inconsistentValue, notFound, syntaxError].

```
GetBooleanValue: PROCEDURE [section, entry: XString.Reader,
    file: NSFile.Reference ← NSFile.nullReference]
    RETURNS [value: BOOLEAN];
```

GetBooleanValue returns the value of a boolean entry. The entry must contain either "TRUE" or "FALSE" or the translated string for TRUE or FALSE as defined in the message files.

```
GetIntegerValue: PROCEDURE [section, entry: XString.Reader,
    index: CARDINAL ← 0, file: NSFile.Reference ← NSFile.nullReference]
    RETURNS [value: LONG INTEGER];
```

GetIntegerValue returns the value of an integer entry. The entry must contain a number that can be parsed by XString.ReaderToNumber. index causes the indexth entry to be read, for repeating entries.

```
GetStringValue: PROCEDURE [section, entry: XString.Reader,
    callBack: PROCEDURE [value: XString.Reader], index: CARDINAL ← 0,
    file: NSFile.Reference ← NSFile.nullReference];
```

GetStringValue calls callBack with the value of a string entry. index causes the indexth entry to be read, for repeating entries.

37.2.2 Current Profiles

ViewPoint supports a current user profile file and a workstation profile file. The current user profile is automatically changed whenever a user logs on or off. The workstation profile contains entries specific to the workstation rather than specific to each user. There is one workstation profile on each workstation.

```
GetUserProfile: PROCEDURE RETURNS [file: NSFile.Reference];
```

GetUserProfile returns the current User profile file. **Note:** Each of the Get and Enumerate procedures uses this file as the file parameter is defaulted.

```
GetWorkstationProfile: PROCEDURE RETURNS [file: NSFile.Reference];
```

GetWorkstationProfile returns the current workstation profile file.

37.2.3 Enumerating a File

EnumerateXXX are useful for applications that look for the same entry in all sections.

```
EnumerateSections: PROCEDURE [callBack: SectionEnumProc,
    file: NSFile.Reference ← NSFile.nullReference];
```

```
SectionEnumProc: TYPE = PROCEDURE [section: XString.Reader]
    RETURNS [stop: BOOLEAN ← FALSE];
```

EnumerateSections calls callBack for each section in file, until stop = TRUE. If file is defaulted, the current user profile is used.

```
EnumerateEntries: PROCEDURE [section: XString.Reader, callBack: EntryEnumProc,
    file: NSFile.Reference ← NSFile.nullReference];
```

```
EntryEnumProc: TYPE = PROCEDURE [entry: XString.Reader]
    RETURNS [stop: BOOLEAN ← FALSE];
```

EnumerateEntries calls callBack for each entry in section in file, until stop = TRUE. If file is defaulted, the current user profile is used.

37.2.4 Errors

```
Error: ERROR [code: ErrorCode];

ErrorCode: TYPE = {invalidParameters, inconsistentValue, notFound, syntaxError};
```

invalidParameters	such as passing in a NIL string.
inconsistentValue	calling GetBooleanValue for an entry that does not have TRUE or FALSE as its value, or calling GetIntegerValue for an entry that will not parse as number.
notFound	asking for an entry that is not in the file.
invalidFile	reading from a file that is not a profile file.
syntaxError	garbage in the file.

```
NotAProfileFile: SIGNAL;
```

The passed file is not a profile file; it has the wrong file type. RESUMEing will read the file anyway.

37.3 Usage/Examples

```
-- In global frame
displayMessage: BOOLEAN ← TRUE;
whereToDisplay: SampleBWSApplicationOps.WhereToDisplay ← window;
messageToDisplay: XString.Reader ← NIL;

-- Called from initialization code
GetOptionsAtLogon: PROCEDURE = {
   logon: Atom.ATOM ← Atom.MakeAtom["Logon"L];
   desktopRef: NSFile.Reference;
   [] ← Event.AddDependency [agent: LogonEvent, myData: NIL, event: logon];
   IF (desktopRef ← StarDesktop.GetCurrentDesktopFile []) # NSFile.nullReference THEN {
      -- If the desktop is NOT null, then a user's already logged on,
      --i.e., we got loaded after logon.
      -- So we go read the options immediately by calling our
      --Event.AgentProcedure directly. --
      desktop: NSFile.Handle ← NSFile.OpenByReference [desktopRef];
      [] ← LogonEvent [event: logon, eventData: LOOPHOLE [desktop], myData: NIL];
      NSFile.Close [desktop];
      };
   };

LogonEvent: Event.AgentProcedure = {
   <<[event: Event.EventType, eventData: LONG POINTER,
   myData: LONG POINTER]
   RETURNS [remove: BOOLEAN ← FALSE, veto: BOOLEAN ← FALSE]>>
   OPEN Ops: SampleBWSApplicationOps;
   mh: XMessage.Handle = Ops.GetMessageHandle[];
CopyMessageToDisplay: PROCEDURE [value: XString.Reader] = {
    messageToDisplay ← XString.CopyReader [value, sysZ]};

GetWhereToDisplay: PROCEDURE [value: XString.Reader] = {
    window: XString.ReaderBody ← XMessage.Get [mh, Ops.kwindow];
    attention: XString.ReaderBody ← XMessage.Get [mh, Ops.kattention];
    both: XString.ReaderBody ← XMessage.Get [mh, Ops.kboth];
    whereToDisplay ← SELECT TRUE FROM
        XString.Equivalent [value, @window] = > window,
        XString.Equivalent [value, @attention] = > attention,
        XString.Equivalent [value, @both] = > both,
        ENDCASE = > window;
    };

section: XString.ReaderBody ← XMessage.Get [mh, Ops.kApplicationName];

entry: XString.ReaderBody ← XMessage.Get [mh, Ops.kDisplayMessage];
displayMessage ← OptionFile.GetBooleanValue [@section, @entry !
    OptionFile.Error = > CONTINUE];

entry ← XMessage.Get [mh, Ops.kMessageToDisplay];
OptionFile.GetStringValue [@section, @entry, CopyMessageToDisplay !
    OptionFile.Error = > CONTINUE];

entry ← XMessage.Get [mh, Ops.kWhereToDisplay];
OptionFile.GetStringValue [@section, @entry, GetWhereToDisplay !
    OptionFile.Error = > CONTINUE];
};
```

37.4 Index of Interface Items

38

PopupMenu

38.1 Overview

The **PopupMenu** interface provides a single procedure that posts a pop-up menu.

38.2 Interface Items

```
Popup: PROCEDURE [
   menu: MenuData.MenuHandle,
   clients:Window.Handle,
   showTitle: BOOLEAN ← TRUE,
   place: Window.Place ← [-1,-1] ];
```

This procedure causes the display of the client's **menu** at or near the indicated **place** in the **rootWindow**; if the **place** [-1,-1] is given, the current cursor position is used. If the point button goes up while the cursor is over one of the menu items, then that item's **MenuData.MenuProc** is called. **clients** is passed to the **MenuData.MenuProc** as the **window** parameter. The **showTitle** field indicates whether the menu's title should be displayed above its command strings.

The implementation assumes that the point button is down; consequently, the menu is displayed until the point button goes up. **Popup** does *not* return until the menu is taken down, regardless of whether a menu item is selected.

38.3 Usage/Examples

Much of the complication in using the **PopupMenu** interface stems from its reliance on **MenuData**. A thorough understanding of how to create a menu is needed before using this interface (see the **MenuData** chapter for details).

38.3.1 Example

```
-- Create the menu:
myMenu: MenuData.MenuHandle ← MenuData.CreateMenu [
    -- ...-- Pass in miscellaneous parameters; see the MenuData interface for details -- ];

PopupMenu.Popup[
    menu: myMenu,
    clients: currentWindow];
    -- showTitle and place are defaulted in this call.
```

38.4 Index of Interface Items

39

ProductFactoring

39.1 Overview

ProductFactoring allows an application to determine whether the customer has purchased a particular application. **ProductFactoring** maintains a record of the applications that have been purchased (enabled) on the workstation's disk. Tools are provided to customers for enabling various applications (options). The enabling of an application is outside the scope of this interface.

ProductFactoring also allows an application to register a name for its product option, thus allowing the product factoring tools to display meaningful names to their users.

39.2 Interface Items

39.2.1 Products and ProductOptions

Product: TYPE = CARDINAL [0..16);

A **Product** refers to a large set of software. (Also see the **ProductFactoringProducts** interface.)

ProductOption: TYPE = CARDINAL [0..28);

A **ProductOption** refers to a particular piece of software that a customer can buy within a **Product**, such as Spreadsheets, Advanced Star Graphics, or Print Service. To obtain a **ProductOption** for a particular application, see your Xerox Sales Represenative.

Option: TYPE = RECORD [product: Product, productOption: ProductOption];

nullOption: Option = . . . ;

An **Option** uniquely identifies a **ProductOption** within a **Product**.

39.2.2 Checking for an Enable Option

Enabled: PROCEDURE [option: Option] RETURNS [enabled: BOOLEAN];

Enabled returns TRUE if option is enabled on this workstation, otherwise FALSE. Typically, an application calls Enabled every time it is called to perform some user operation such as opening an icon. Enabled is fast; it does not read the file every time it is called. It may raise Error[notStarted] if there is no product factoring file on the workstation.

39.2.3 Describing a Product and an Option

```
DescribeProduct: PROCEDURE [product: Product, desc: XString.Reader];
```

Provides a name for product. desc is copied to an internal zone. May raise Error[illegalProduct] if the value of product is out of range.

```
DescribeOption: PROCEDURE [option: Option, desc: XString.Reader,
  prerequisite: Prerequisite ← nullPrerequisite];

Prerequisite: TYPE = RECORD [
    prerequisiteSpec: BOOLEAN ← FALSE,
    option: Option];

nullPrerequisite: Prerequisite = [FALSE, nullOption];
```

Describes option. desc is a name for the option. prerequisite specifies any other options that this option depends on. All data is copied to an internal zone. Use of this procedure overrides any earlier definition with the same option value. May raise Error[illegalProduct] if the value of option.product is out of range. May raise Error[illegalOption] if the value of option.productOption is out of range. May raise Error[missingProduct] if option.product has not yet been defined.

39.2.4 Errors

```
Error: ERROR [type: ErrorType];

ErrorType: TYPE = {
    dataNotFound, notStarted, illegalProduct, illegalOption,
    missingProduct, missingOption};
```

dataNotFound	The product data file is missing.
notStarted	Start proc has not been called yet.
illegalProduct	Not a legal Product value.
illegalOption	Not a legal ProductOption value.
missingProduct	The Product specified has not yet been defined.
missingOption	The ProductOption specified has not yet been defined.

39.3 Usage/Examples

```
-- In global frame --
sampleApplicationPFOption: ProductFactoring.ProductOption = 27;
```

```
    -- 27 was chosen arbitrarily for this sample. --
    -- A real application should obtain a real ProductOption! --

-- Called during initialization --
InitProductFactoring: PROCEDURE = {
    mh: XMessage.Handle = SampleBWSApplicationOps.GetMessageHandle[];
    rb: XString.ReaderBody ← XMessage.Get [mh,
        SampleBWSApplicationOps.kApplicationName];
    ProductFactoring.DescribeOption [
        option: [ product: ProductFactoringProducts.Star,
            productOption: sampleApplicationPFOption],
        desc: @rb];
    };

-- GenericProc --
GenericProc: Containee.GenericProc = {
    IF ~ProductFactoring.Enabled [option: [
            product: ProductFactoringProducts.Star,
            productOption: sampleApplicationPFOption]] THEN {
        mh: XMessage.Handle ← SampleBWSApplicationOps.GetMessageHandle[];
        rb: XString.ReaderBody ← XMessage.Get [mh, SampleBWSApplicationOps.kNotEnabled];
        ERROR Containee.Error [@rb];
        };
    SELECT atom FROM
    . . .
    };
```

39.4 Index of Interface Items

Item	Page
DescribeProduct: PROCEDURE	2
DescribeOption:PROCEDURE	2
Enabled: PROCEDURE	1
Error: ERROR	2
ErrorType: TYPE	2
nullOption: Option	1
nullPrerequisite: Prerequisite	2
Option: TYPE	1
Prerequisite: TYPE	2
Product: TYPE	1
ProductOption: TYPE	1

40

ProductFactoringProducts

40.1 Overview

ProductFactoringProducts defines the **ProductFactoring.Products** for various Xerox products. (See the **ProductFactoring** interface).

40.2 Interface Items

Product: TYPE = ProductFactoring.Product;

Star: Product = 0;

Star defines the Xerox Star (aka ViewPoint) workstation product.

Services: Product = 1;

Services defines the Xerox network services product.

DFonts: Product = 3;

DFonts defines the product for Xerox display fonts.

PFonts: Product = 4;

PFonts defines the product for Xerox printer fonts.

ViewPoint: Product = 5;

ViewPoint defines a product for Xerox ViewPoint applications.

ViewPointApps: Product = 6;

ViewPointApps defines a product for Xerox ViewPoint applications.

Converter: Product = 7;

Converter defines the product for Xerox ViewPoint converter icon applications.

Services2: Product = 8;

Services2 defines another Xerox network services product. It is defined in ProductFactoringProductsExtras.mesa.

Languages: Product = 9;

Languages defines the Xerox languages product. It is defined in ProductFactoringProductsExtras2.mesa.

FujiUnique: Product = 10;

Fuji2Unique: Product = 11;

FujiUnique and **Fuji2Unique** define Fuji Xerox products. These are defined in ProductFactoringProductsExtras3.mesa.

OahuUnique: Product = 12;

OahuUnique defines the Xerox Oahu product. It is defined in ProductFactoringProductsExtras4.mesa.

RankUnique: Product = 13;

RankUnique defines the Rank Xerox product. It is defined in ProductFactoringProductsExtras4.mesa.

40.3 Index of Interface Items

41

PropertySheet

41.1 Overview

The **PropertySheet** interface allows clients to create property sheets. A property sheet shows the user the properties of an object and allows the user to change these properties. Several different types of properties are supported. The most common ones are boolean, choice (enumerated), and text. (See Figure 41.1.)

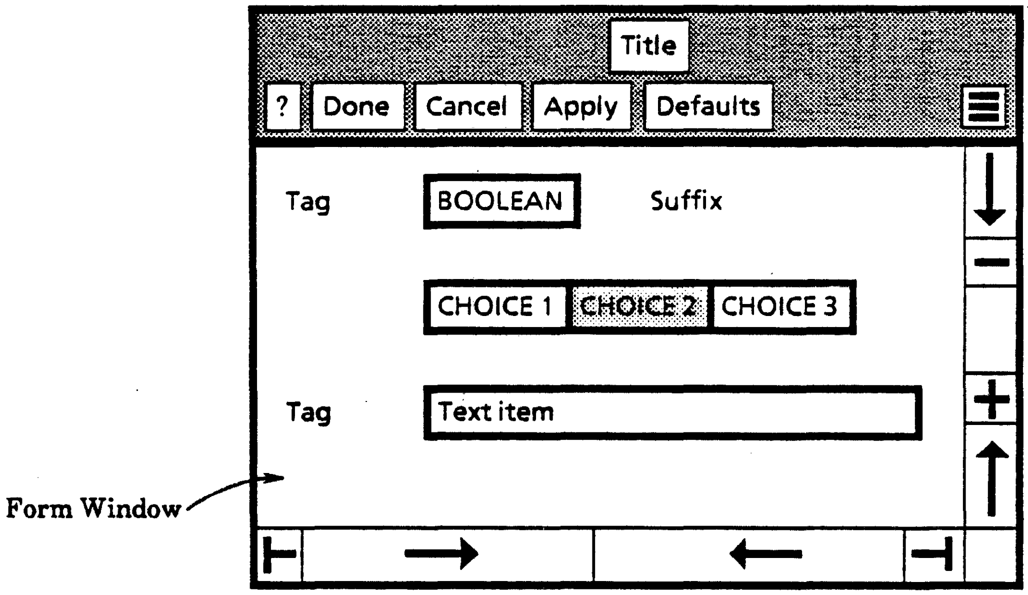


Figure 41.1. A Property Sheet

From a client's point of view, a property sheet is a Star window shell with a form window as a body window. See the **StarWindowShell** and **FormWindow** interfaces. The **FormWindow** interface especially must be understood in order to create a property sheet.

A property sheet is created by calling **PropertySheet.Create**, providing a procedure that will make the form items in the form window (a **FormWindow.MakeItemsProc**), a list of commands to put in the header of the property sheet, such as *Done*, *Cancel*, and *Apply* (**PropertySheet.MenuItems**), and a procedure to call when the user selects one of these commands (a **PropertySheet.MenuItemProc**). **PropertySheet.Create** returns the **StarWindowShell.Handle** for the property sheet. When the user selects one of the commands in the header of the property sheet, the client's PropertySheet.MenuItemProc is called. If the user selected *Done* , for example, the client can then verify and apply any changes the user made to the object's properties.

PropertySheet also provides the capability to create linked property sheets. Several property sheets may be logically linked together in the same property sheet shell. This is accomplished by changing form windows within a property sheet's Star window shell; and by having an additional choice item that specifies which form window is currently displayed. Linked property sheets are further described in the section on Linked Property Sheets below.

41.2 Interface Items

41.2.1 Create a PropertySheet (Not a Linked One)

```
Create: PROCEDURE [
   formWindowItems: FormWindow.MakeItemsProc,
   menuItemProc: MenuItemProc,
   size: Window.Dims,
   menuItems: MenuItems ← propertySheetDefaultMenu,
   title: XString.Reader ← NIL,
   placeToDisplay: Window.Place ← nullPlace,
   formWindowItemsLayout:FormWindow.LayoutProc ← NIL,
   windowAttachedTo: StarWindowShell.Handle ← [NIL],
   globalChangeProc: FormWindow.GlobalChangeProc ← NIL,
   display: BOOLEAN ← TRUE,
   clientData: LONG POINTER ← NIL,
   afterTakenDown: MenuItemProc ← NIL,
   zone: UNCOUNTED ZONE ← NIL]
   RETURNS [shell: StarWindowShell.Handle];
```

Create creates a property sheet.

formWindowItems is a client-supplied procedure that is passed a body window of the property sheet. It should fill the window with the form items that make up the main body of the property sheet. (See the **FormWindow** interface for a full description of how to create form items in a window.)

menuItemProc is a client-supplied procedure that is called whenever the user selects one of the menu items in the header of the property sheet window. (See §41.2.2 below.)

size is the preferred size of the property sheet star window shell.

menuItems specifies the menu items that are displayed in the header of the property sheet. The default is *? (help)*, *Done*, and *Cancel*.

title is the title to be displayed in the header of the property sheet.

placeToDisplay is the preferred location on the screen of the property sheet. If the default is taken, **Create** will calculate the place to display.

formWindowItemsLayout specifies the desired position of the form items in the **FormWindow**. (See FormWindow.LayoutProc for a full description.). If formWindowItemsLayout is NIL, then FormWindow.DefaultLayout of one item per line is used

windowAttachedTo is the StarWindowShell that this property sheet is showing properties for. If windowAttachedTo is not NIL, then the user will not be able to close windowAttachedTo until this property sheet is closed. (See also StarWindowShell.Create.host.)

globalChangeProc is called if any item in the main form window is changed. (See FormWindow.GlobalChangeProc for a full description).

display indicates whether the property sheet should actually be displayed on the screen (inserted into the visible window tree) or just created but not actually painted on the screen (not inserted into the visible window tree). If this is a property sheet for a file (i.e., if it is being created as the result of a call to a Containee.GenericProc [atom: Props]), then display should be FALSE and the StarWindowShell.Handle should be returned from the GenericProc so that, for example, the desktop implementation can put the property sheet on the display by calling StarWindowShell.Push.

clientData will be passed to formWindowItems, formWindowItemsLayout, and menuItemProc. Fine Point: formWindowItems will not be called after Create returns and therefore may be nested.

The afterTakenDown is called after the property sheet has been removed from the screen. The return parameter of the MenuItemProc is ignored in this case. Note: Clients must still provide a regular MenuItemProc.

Clients may pass in a zone to be used instead of the default zone created by the StarWindowShell implementation.

shell is the property sheet. shell will be a bitmap-under window if storage space is available. Clients who desire a property sheet without a bitmap-under should use CreateWithBitmapUnderOption. The typical property sheet should be a bitmap-under window since bitmap-under windows are moved and deleted faster than non bitmap-under windows.

```
nullPlace: Window.Place;
```

nullPlace defines the default for placement of a property sheet. If the default is used, the property sheet is placed at an appropriate place on the screen.

```
CreateWithBitmapUnderOption: PROCEDURE [
  formWindowItems: FormWindow.MakeItemsProc,
  menuItemProc: MenuItemProc,
  size: Window.Dims,
  menuItems: MenuItems ← propertySheetDefaultMenu,
  title: XString.Reader ← NIL,
  placeToDisplay: Window.Place ← nullPlace,
  formWindowItemsLayout:FormWindow.LayoutProc ← NIL,
  windowAttachedTo: StarWindowShell.Handle ← [NIL],
  globalChangeProc: FormWindow.GlobalChangeProc ← NIL,
  display: BOOLEAN ← TRUE,
  clientData: LONG POINTER ← NIL,
  afterTakenDown: MenuItemProc ← NIL,
  bitmapOption: StarWindowShellExtra5.BitmapUnderOption ← noBitmapUnder,
zone: UNCOUNTED ZONE ← NIL]
RETURNS [shell: StarWindowShell.Handle];
```

CreateWithBitmapUnderOption is identical to **Create** except for the additional parameter **bitmapOption**. If **bitmapOption** is **noBitmapUnder**, no attempt is made to make the property sheet a bitmap-under. If **bitmapOption** is **bitmapUnder** or **maybeBitmapUnder**, the property sheet will be a bitmap-under window if enough storage space is available. Clients must use **CreateWithBitmapUnderOption** to create a property sheet **without** a bitmap-under since a property sheet created through **Create** is automatically a bitmap-under window if storage space is available. Clients of **CreateWithBitmapUnderOption** need not be concerned with managing the bitmap-under storage. Fine Point: This procedure is currently exported through **PropertySheetExtra.**

41.2.2 Menu Items and the MenuItemProc

```
MenuItemType: TYPE = {done, apply, cancel, defaults, start, reset};

MenuItems: TYPE = PACKED ARRAY MenuItemType OF BooleanFalseDefault;

BooleanFalseDefault: TYPE = BOOLEAN ← FALSE;

propertySheetDefaultMenu: MenuItems = [done: TRUE, apply: TRUE, cancel: TRUE];

optionSheetDefaultMenu: MenuItems = [start: TRUE, cancel: TRUE];
```

The client specifies a set of commands to be placed in the header of the property sheet. **MenuItemType** specifies all of the possible commands. **MenuItems** specifies a set of these commands and is passed to **PropertySheet.Create**. **propertySheetDefaultMenu** and **optionSheetDefaultMenu** specify two common sets of commands.

```
MenuItemProc: TYPE = PROCEDURE [
  shell: StarWindowShell.Handle,
  formWindow: Window.Handle,
  menuItem: MenuItemType]
  RETURNS [ok: BOOLEAN ← FALSE];
```

The client supplies a **MenuItemProc** when a property sheet is created. It is called whenever the user selects one of the menu items in the header of the property sheet. **formWindow** is the main form window of the property sheet. **menuItem** is the type of menu item that the user selected. The client typically (when the user selects *Done* or *Apply*) retrieves the values of the items that the user edited (using **FormWindow.HasChanged** and **FormWindow.GetXXXItemValue** procedures), verifies that the values are meaningful (for example, numbers that are within proper range), and applies the new values to the properties of the object this property sheet represents.

The return parameter **ok** has slightly different meanings in the following two cases:

1. For an ordinary property sheet (not a linked property sheet), the **MenuItemProc** is called when the user selects a command and the return parameter indicates whether the property sheet should be destroyed.
2. For a linked property sheet, the **MenuItemProc** is called both when the user selects a command in the header (in which the case above applies) and when the client calls

SwapExistingFormWindows or **SwapFormWindows** with **apply** = TRUE. In this case the **MenuItemProc** is called to allow the client to apply any changes made to the form window sheet being linked from. The **menuItem** parameter will be "done"; the return parameter indicates whether to allow the swap to actually occur. **ok** = FALSE indicates that there is something invalid in the form window and the client does not want the swap to occur (the client typically posts a message before returning). If **ok** = TRUE, the swap occurs.

Note: The client need not worry about these cases when writing the **MenuItemProc**, but can simply write the "done" code as usual. If the user selects *Done* and the **MenuItemProc** returns **ok** = TRUE, the property sheet is destroyed. If the user links to another sheet on a linked property sheet and the **MenuItemProc** returns **ok** = TRUE, the sheets are swapped, rather than the whole property sheet being destroyed.

41.2.3 Linked PropertySheets

Several property sheets may be logically linked together in the same property sheet. This is accomplished by changing form windows within a property sheet's Star window shell, and having an additional choice item that specifies which form window is currently displayed. See Figure 41.2 below.

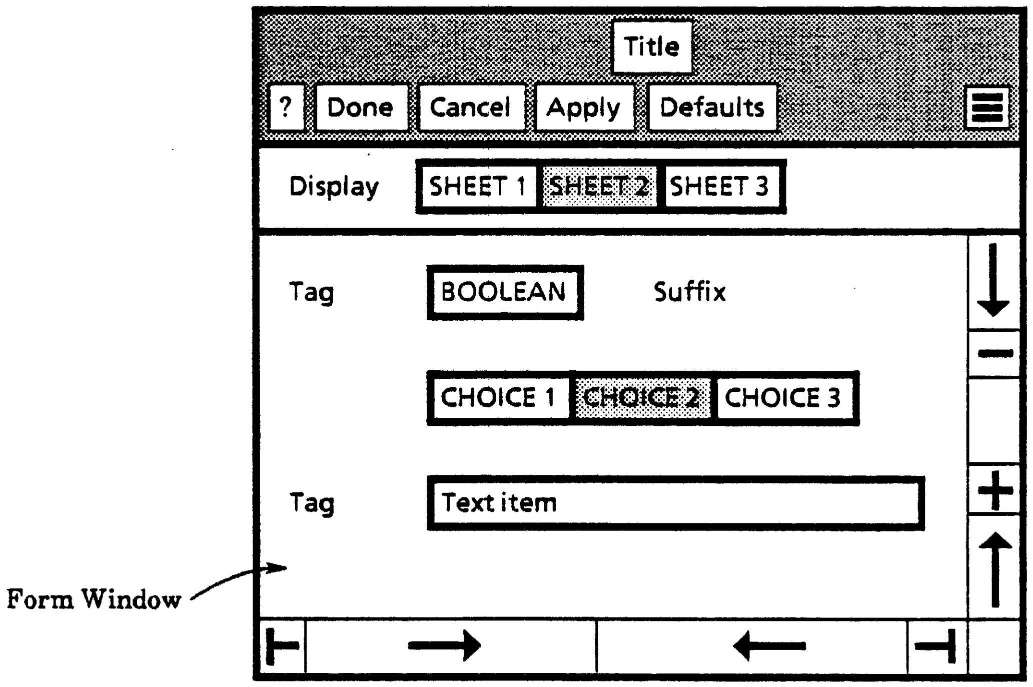


Figure 41.2 A Linked Property Sheet

This additional choice item actually resides in an additional form window, called a *link window*. This link window is another body window of the Star window shell. The link window remains visible all the time, while the main form window may be swapped. The client does this by supplying a FormWindow.ChoiceChangeProc for the single choice item in the link window. Then when the user selects a new choice for that item, the client (in the ChoiceChangeProc) calls **SwapFormWindows** or **SwapExistingFormWindows** to change the main form window. **Note:** *Only one* main form window is installed in the Star window shell at a time. A linked property sheet is created by calling **CreateLinked**.

```
CreateLinked: PROCEDURE [
   formWindowItems: FormWindow.MakeItemsProc,
   menuItemProc: MenuItemProc,
   size: Window.Dims,
   menuItems: MenuItems ← propertySheetDefaultMenu,
   title: XString.Reader ← NIL,
   placeToDisplay: Window.Place ← nullPlace,
   formWindowItemsLayout:FormWindow.LayoutProc ← NIL,
   windowAttachedTo: StarWindowShell.Handle ← [NIL],
   globalChangeProc: FormWindow.GlobalChangeProc ← NIL,
   display: BOOLEAN ← TRUE,
   linkWindowItems: FormWindow.MakeItemsProc,
   linkWindowItemsLayout:FormWindow.LayoutProc ← NIL,
   clientData: LONG POINTER ← NIL,
   afterTakenDownProc: MenuItemProc ← NIL,
   zone: UNCOUNTED ZONE ← NIL]
   RETURNS [shell: StarWindowShell.Handle];
```

CreateLinked creates a linked property sheet. Creating a linked property sheet is almost identical to creating an ordinary property sheet, (see **Create** above for a full description of all the parameters), except **CreateLinked** has the additional parameters linkWindowItems and linkWindowItemsLayout. linkWindowItems is called to make the choice item in the link window. It should create a single choice item with a FormWindow.ChoiceChangeProc. linkWindowItemsLayout is called to specify the position of the choice item in the link window. The default places the item appropriately in the link window, so most clients will want to take the default for linkWindowItemsLayout. **Note:** formWindowItems and formWindowItemsLayout specify the main form window that is *initially* visible in the property sheet.

```
CreateLinkedWithBitmapUnderOption: PROCEDURE [
   formWindowItems: FormWindow.MakeItemsProc,
   menuItemProc: MenuItemProc,
   size: Window.Dims,
   menuItems: MenuItems ← propertySheetDefaultMenu,
   title: XString.Reader ← NIL,
   placeToDisplay: Window.Place ← nullPlace,
   formWindowItemsLayout:FormWindow.LayoutProc ← NIL,
   windowAttachedTo: StarWindowShell.Handle ← [NIL],
   globalChangeProc: FormWindow.GlobalChangeProc ← NIL,
   display: BOOLEAN ← TRUE,
   linkWindowItems: FormWindow.MakeItemsProc,
   linkWindowItemsLayout:FormWindow.LayoutProc ← NIL,
   clientData: LONG POINTER ← NIL,
   afterTakenDownProc: MenuItemProc ← NIL,
   bitmapOption: StarWindowShellExtra5.BitmapUnderOption ← noBitmapUnder,
   zone: UNCOUNTED ZONE ← NIL]
   RETURNS [shell: StarWindowShell.Handle];
```

CreateLinkedWithBitmapUnderOption is identical to **CreateLinked** except for the additional parameter **bitmapOption**. See **CreateWithBitmapUnderOption** for an explanation of the **bitmapOption** parameter. Fine Point: This procedure is currently exported through **PropertySheetExtra**.

```
SwapFormWindows: PROCEDURE [
   shell: StarWindowShell.Handle,
   newFormWindowItems: FormWindow.MakeItemsProc,
   newFormWindowItemsLayout:FormWindow.LayoutProc ← NIL,
   apply: BOOLEAN ← TRUE,
   destroyOld: BOOLEAN ← TRUE,
   newMenuItemProc: MenuItemProc ← NIL,
   newMenuItems: MenuItems ← ALL[FALSE],
   newTitle: XString.Reader ← NIL,
   newGlobalChangeProc: FormWindow.GlobalChangeProc ← NIL,
   newAfterTakenDownProc: MenuItemProc ← NIL]
   RETURNS [old: Window.Handle];
```

SwapFormWindows swaps the main form window of a property sheet with a new one. This will usually be called from the **FormWindow.ChoiceChangeProc** of the choice item in the link window. May raise **Error [notAPropSheet]**.

shell is the property sheet.

newFormWindowItems supplies the items for the new window.

newFormWindowItemsLayout specifies the layout for the items in the new form window.

apply specifies whether any changes to the current form window should be applied before the swap. If **apply** = TRUE, the current **MenuItemProc** for shell is called with **menuItem** = **apply**. If **apply** = FALSE, the **MenuItemProc** is not called.

The **destroyOld** parameter indicates whether the old form window should be destroyed or not. If **destroyOld** = FALSE, then the return parameter is the old form window, else the return parameter is NIL. This allows clients to perform the following typical sequence of events:

1. Create a linked property sheet using **CreateLinked**.
2. The first time the user links to another sheet, call **SwapFormWindows** with **destroyOld** = FALSE and save the old form window.
3. When the user goes back to the first sheet, call **SwapExistingFormWindows**, supplying the previously saved old form window, and thus avoiding having to create the first form window again.

newMenuItemProc allows the client to install a different **MenuItemProc** than the one that was supplied with **CreateLinked**.

newAfterTakenDownProc allows the client to install a different takedown **MenuItemProc** than the one that was supplied with **CreateLinked**.

newMenuItems, newTitle, and newGlobalChangeProc allow the client to change these as well.

If the default newMenuItemProc, newMenuItems, newTitle, or newGlobalChangeProc is taken, the current values are retained.

```
SwapExistingFormWindows: PROCEDURE [
   shell: StarWindowShell.Handle,
   new: Window.Handle,
   apply: BOOLEAN ← TRUE,
   newMenuItemProc: MenuItemProc ← NIL,
   newMenuItems: MenuItems ← ALL[FALSE],
   newTitle: XString.Reader ← NIL,
   newAfterTakenDownProc: MenuItemProc ← NIL]
   RETURNS [old: Window.Handle];
```

SwapExistingFormWindows swaps the main form window of a property sheet with a new one. The new form window must already exist. If it does not, use SwapFormWindow. new is the new form window. apply, newMenuItemProc, newMenuItems, and newTitle are the same as in SwapFormWindow. old is the previous main form window. May raise Error [notAPropSheet].

41.2.4 Miscellaneous

```
GetFormWindows: PROCEDURE [shell: StarWindowShell.Handle]
   RETURNS [form, link: Window.Handle];
```

GetFormWindows returns the current form windows of shell. If shell is not a linked property sheet, link is NIL. May raise Error [notAPropSheet].

```
InstallFormWindow: PROCEDURE [
   shell: StarWindowShell.Handle,
   menuItemProc: MenuItemProc,
   menuItems: MenuItems ← propertySheetDefaultMenu,
   title: XString.Reader ← NIL,
   formWindow: Window.Handle,
   afterTakenDownProc: MenuItemProc ← NIL];
```

InstallFormWindow installs formWindow in shell. May raise Error [notAPropSheet].

41.2.5 Signals and Errors

```
Error: ERROR [code: ErrorCode];
```

```
ErrorCode: TYPE = {notAPropSheet};
```

Error [notAPropSheet] is raised if a StarWindowShell.Handle that is not a property sheet is passed to a PropertySheet procedure .

41.3 Usage/Examples

41.3.1 Flow Description of Creating a Property Sheet

The following describes the sequence of calls involved in creating and taking down a property sheet, including ViewPoint interfaces and clients.

1. Client calls PropertySheet.Create, supplying a FormWindow.MakeItemsProc, a FormWindow.LayoutProc (optional), and a PropertySheet.MenuItemProc.

2. PropertySheet.Create creates a Star window shell and a body window inside the StarWindowShell. It then calls FormWindow.Create, passing in the body window.

3. FormWindow.Create calls the client's FormWindow.MakeItemsProc.

4. The client's FormWindow.MakeItemsProc creates the items in the property sheet by calling various FormWindow.MakeXXXItem procedures.

5. FormWindow.Create calls the client's FormWindow.LayoutProc. If the client did not provide one, a default LayoutProc provided by FormWindow is called.

6. The FormWindow.LayoutProc makes calls to FormWindow.AppendLine and FormWindow.AppendItem to specify the layout of the items created by the FormWindow.MakeItemsProc.

7. FormWindow.Create returns to PropertySheet.Create. PropertySheet.Create returns to the client. The client returns to the notifier process.

8. The property sheet is now on the screen and the notifier process is waiting for the user.

9. The user changes some values in the property sheet. This is all managed by FormWindow; the client gets called only if there is a FormWindow.BooleanChangeProc or FormWindow.ChoiceChangeProc or FormWindow.GlobalChangeProc.

10. The user selects *Done* in the header of the property sheet.

11. A procedure inside of PropertySheet is called. PropertySheet calls the client's PropertySheet.MenuItemProc.

12. The client's PropertySheet.MenuItemProc checks for any changed values (FormWindow.HasBeenChanged and FormWindow.HasAnyBeenChanged) and calls the appropriate FormWindow.GetXXXItemValue to obtain the new values. The client validates and applies these new values, then returns an indication of whether the property sheet should be taken down.

13. PropertySheet takes down the property sheet and returns to the notifier.

14. END.

41.3.2 An Ordinary Property Sheet

This example creates a property sheet from some arbitrary properties and then applies the user's changes to those properties. It uses a rather contrived set of properties described by **Properties** and **PropertiesObject**. In general, a real property sheet would get its properties from some real object. This example will produce the property sheet shown in Figure 41.1.

```
-- PropertySheetExample.mesa

DIRECTORY
   FormWindow USING [
      ChoiceItem, GetBooleanItemValue, GetChoiceItemValue, GetTextItemValue,
      HasAnyBeenChanged, HasBeenChanged, ItemKey, MakeBooleanItem,
      MakeChoiceItem, MakeItemsProc, MakeTextItem, SetBooleanItemValue,
      SetChoiceItemValue, SetTextItemValue],
   PropertySheet USING [Create, MenuItemProc],
   StarWindowShell USING [Handle],
   XString USING [FreeReaderBytes, FromSTRING, ReaderBody],
   Window USING [Handle];

PropertySheetExample: PROGRAM IMPORTS FormWindow, PropertySheet, XString = {

Properties: TYPE = LONG POINTER TO PropertiesObject;

PropertiesObject: TYPE = RECORD [
   boolean: BOOLEAN,
   choice: Choices,
   text: XString.ReaderBody];

Items: TYPE = {boolean, choice, text};

Choices: TYPE = {choice1, choice2, choice3};

zone: UNCOUNTED ZONE ← . . . ;

MakePropertySheet: PROCEDURE [props: Properties]
   RETURNS [shell: StarWindowShell.Handle] = {
   title: XString.ReaderBody ← XString.FromSTRING ["Title"L];

   shell ← PropertySheet.Create [
      formWindowItems: MakeItems,
      menuItemProc: MenuItemProc,
      menuItems: [help: TRUE, done: TRUE, cancel: TRUE,
         apply: TRUE, defaults: TRUE],
      size: [w: 300, h: 200],
      title: @title,
      clientData: props];
   };
```

```
MakeItems: FormWindow.MakeItemsProc = {
  props: Properties ← clientData;
  tag: XString.ReaderBody ← XString.FromSTRING["Tag"L];

  BEGIN
  label: XString.ReaderBody ← XString.FromSTRING["BOOLEAN"L];
  suffix: XString.ReaderBody ← XString.FromSTRING["suffix"L];
  FormWindow.MakeBooleanItem [
    window: window,
    myKey: Items.boolean.ORD,
    tag: @tag,
    suffix: @suffix,
    label: [string [label] ],
    initBoolean: props.boolean ];
  END;

  BEGIN
  c1: XString.ReaderBody ← XString.FromSTRING["CHOICE 1"L];
  c2: XString.ReaderBody ← XString.FromSTRING["CHOICE 2"L];
  c3: XString.ReaderBody ← XString.FromSTRING["CHOICE 3"L];
  choices: ARRAY [0..3) OF FormWindow.ChoiceItem ← [
    [string[Choices.choice1.ORD, c1] ],
    [string[Choices.choice2.ORD, c2] ],
    [string[Choices.choice3.ORD, c3] ] ];
  FormWindow.MakeChoiceItem [
    window: window,
    myKey: Items.choice.ORD,
    values: DESCRIPTOR[choices],
    initChoice: props.choice.ORD ];
  END;

  FormWindow.MakeTextItem [
    window: window,
    myKey: Items.text.ORD,
    tag: @tag,
    width: 40,
    initString: @props.text ];

};

MenuItemProc: PropertySheet.MenuItemProc = {
  props: Properties ← clientData;
  SELECT menuItem FROM
    help = > . . . ;
    done = > RETURN[destroy: ApplyAnyChanges[formWindow, props].ok];
    cancel = > RETURN[destroy: TRUE];
    apply = > [] ← ApplyAnyChanges[formWindow, props];
    defaults = > SetDefaults[formWindow, props];
    ENDCASE = > ERROR;
  RETURN[destroy: FALSE];
  };
```

```
ApplyAnyChanges: PROC [window: Window.Handle, props: Properties]
   RETURNS [ok: BOOLEAN] = BEGIN
   IF ~FormWindow.HasAnyBeenChanged [window] THEN RETURN [ok: TRUE];
   FOR eachItem: Items IN Items DO
      itemKey: FormWindow.ItemKey = eachItem.ORD;
      IF ~FormWindow.HasBeenChanged [window, itemKey] THEN LOOP;
      SELECT eachItem FROM
         boolean = > props.boolean ← FormWindow.GetBooleanItemValue[window, itemKey];
         choice = > props.choice ← VAL[ FormWindow.GetChoiceItemValue[window, itemKey] ];
         text = > {
              XString.FreeReaderBytes [r: @props.text, z: zone];
              props.text ← FormWindow.GetTextItemValue [window, itemKey, zone]};
         ENDCASE;
      ENDLOOP;
   RETURN [ok: TRUE];
   END;-- ApplyAnyChanges

SetDefaults: PROC [window: Window.Handle, props: Properties] =
   BEGIN
   defaultText: XString.ReaderBody ← XString.FromSTRING["Text item"L];
   FormWindow.SetBooleanItemValue [
      window: window,
      item: Items.boolean.ORD,
      newValue: FALSE ];
   FormWindow.SetChoiceItemValue [
      window: window,
      item: Items.choice.ORD,
      newValue: Choices.choice2.ORD ];
   FormWindow.SetTextItemValue [
      window: window,
      item: Items.text.ORD,
      newValue: @defaultText];
   END;

}...
```

41.4 Index of Interface Items

42 Prototype

42.1 Overview

Prototype manipulates prototype files. A *prototype file* is a blank copy of an application's file that the user can copy. Prototype files are in the Directory icon under "Blank Documents, Folders, etc."

A prototype file resides in the prototype catalog (see the **Catalog** interface) and is uniquely identified by it is file type, subtype, and version. The subtype distinguishes between objects of the same file type, such as the blank document and the basic graphics transfer document. Subtype is stored in an extended attribute on the prototype file. A nonexistent subtype is equivalent to subtype 0.

Version is stored in the BWS-standard version extended attribute (see **BWSAttributeTypes**). The intent is that clients need only examine the version to determine if the prototype is current. A nonexistent version attribute is equivalent to version 0.

Prototype provides **Find** and **Create** procedures. A client typically calls **Find** and if it returns **NSFile.nullReference**, then call **Create**.

42.2 Interface Items

```
Version: TYPE = CARDINAL;

Subtype: TYPE = CARDINAL;

Find: PROCEDURE [type: NSFile.Type, version: Version,
    subtype: Subtype ← 0, session: NSFile.Session ← NSFile.nullSession]
    RETURNS [reference: NSFile.Reference];
```

Find returns a reference for the file with the specified **type**, **version**, and **subtype**. If the file does not exist, **NSFile.nullReference** is returned.

```
Create: PROCEDURE [
    name: XString.Reader,
    type: NSFile.Type,
    version: Version,
    subtype: Subtype ← 0,
    size: LONG CARDINAL ← 0,
    isDirectory: BOOLEAN ← FALSE,
    session: NSFile.Session ← NSFile.nullSession]
    RETURNS [prototype: NSFile.Handle];
```

Creates a file in the prototype catalog with the specified **name**, **type**, **version**, **subtype**, **size** in bytes, and **isDirectory** attribute.

```
Add: PROCEDURE [file: NSFile.Handle, version: Version,
    subtype: Subtype ← 0, session: NSFile.Session ← NSFile.nullSession];
```

Moves an already existing **file** into the prototype catalog, assigning it the given **version** and **subtype**. Fine point: This is in PrototypeExtra in ViewPoint.

```
PurgeOldVersions: PROCEDURE [type: NSFile.Type, current: Version, subtype: Subtype ← 0];
```

Deletes any versions of the given prototype that are older (smaller number) than **current**. **PurgeOldVersions** assumes that higher version numbers are more recent than lower version numbers. If this is not true for your version numbers, do not call this operation .

42.3 Usage/Examples

This is an example of a procedure that an application probably calls at initialization time.

```
sampleIconFileType: NSFile.Type = ...;

version: CARDINAL = ...;

FindOrCreateIconFile: PROCEDURE = {
    name: XString.ReaderBody ← XString.FromSTRING["Sample Icon"L];
    -- This name should really come from XMessage.
    IF (Prototype.Find [
        type: sampleIconFileType, version: version] = NSFile.nullReference) THEN
        NSFile.Close [Prototype.Create [
            name: @name, type: sampleIconFileType, version: version] ];
    };
```

42.4 Index of Interface Items

43

Scrollbar

43.1 Overview

Scrollbar provides the means of attaching scrollbars to windows. Clients provide a number of procedures that perform various scrolling actions. For a comprehensive view of all the subwindow interfaces and their intended use, see the Subwindow Overview chapter.

43.2 Interface Items

43.2.1 Attaching Scrollbars

```
Attach: PROCEDURE [
    window: Window.Handle,
    vertical,, horizontal: BOOLEAN ← TRUE,
    single: SingleScrollProc ← NIL,
    jump: JumpScrollProc ← NIL,
    scrollbarInfo: ScrollbarInfoProc ← NIL,
    thumb: ThumbScrollProc ← NIL,
    feedback: ThumbFeedbackProc ← NIL,
    zone: UNCOUNTED ZONE];

Type: TYPE = {horizontal, vertical};
```

Attach will attach a scrollbar of **type** to **window**. Scrollbars are younger siblings of the viewer (**window**) they are attached to. **single** is the proc that will be called when the user invokes scrolling in one of the arrows or paging symbols (+ -). **jump** scrolling is currently not implemented. **thumb** is called when the user points down in the thumbing region. feedback is called for each mouse action to allow the client to provide interim feedback like a sideways arrow. **scrollbarInfo** will be called after **thumb** to get the percentage and offset information needed to paint the thumbing feedback in the scrollbar.

Adjust: PROCEDURE [**window**: Window.**Handle**];

Should be called when **window** is changing size to allow the attached scrollbars to adjust their size accordingly. Should also be called after the call to **Attach** to establish initial size.

Destroy: PROCEDURE [**window**: Window.**Handle**, **type**: **Type**];

Destroys the context associated with the scrollbar of **type** that is attached to **window**.

43.2.2 Scroll proc TYPES and PROCS

In each of the following procedure types, **window** refers to the viewer of the subwindow that the scrollbars are attached to. **type** refers to the specific scrollbar on the viewer, horizontal or vertical.

```
SingleScrollProc: TYPE = PROC [
    window: Window.Handle,
    flavor: SingleScrollFlavor,
    amount: NATURAL,
    arrowScrollAction: ArrowScrollAction ← go,
    type: Type];
```

SingleScrollFlavor: TYPE = {**pageFwd, pageBwd, forward, backward**};

ArrowScrollAction: TYPE = {**start, go, stop**};

defaultSmoothScrollAmount: NATURAL;

A **SingleScrollProc** is the basic scrolling proc that should provide for **forward**, **backward** continuous scrolling, and **pageFwd**, **pageBwd** paging. It is called whenever the user points at an arrow or the plus or minus sign in a scrollbar. Scroll distance will be determined by **amount**. If **flavor** is **forward** or **backward** then **amount** is in screen dots. (**defaultSmoothScrollAmount** can be used by the caller if desired). If **flavor** is **pageFwd** or **pageBwd** then **amount** is in pages. A **SingleScrollProc** is called with **arrowScrollAction** = **start** when the user begins the scrolling action. **SingleScrollProc** is called with **arrowScrollAction** = **go** for each successive call until the user terminates scrolling. At termination **arrowScrollAction** = **stop**. The **SingleScrollProc** will be called repeatedly as long as the user has the mouse button down over one of the arrows, thus producing continuous scrolling.

```
JumpScrollProc: TYPE = PROC [
    window: Window.Handle,
    direction: JumpScrollFlavor,
    percent: Percent],
    type: Type;
```

JumpScrollFlavor: TYPE = {**forward, backward**};

Percent: TYPE = [**0..100**];

The client may provide a **JumpScrollProc** for Tajo-like jump scrolling. **percent** is relative to window. Currently not implemented.

```
ScrollbarInfoProc: TYPE = PROC[
    window: Window.Handle,
    type: Type]
    RETURNS [offset, portion: Percent];
```

The clients **ScrollbarInfoProc** is called to get information to properly paint the thumbing region whenever the user has scrolled to a new position in the window. This is true whether the user is thumbing, paging or continuous scrolling. The client returns the **offset** from the beginning of the object and **portion** (percentage of the entire object) that is visible in the viewer. The scrollbar display will be updated using this information to properly paint the thumbing region (tajo gray bar, star diamond, etc.)

```
ThumbWithin: TYPE = {all, page, other};

ThumbAction: TYPE = {down, track, up};
```

In the following procedures **within** equates **percent** with an extent:
- **all** = > **percent** is relative to the entire object.
- **page** = > **percent** is relative to a client defined page.
- **other** = > is currently unused.

```
ThumbFeedbackProc: TYPE = PROC[
    window: Window.Handle,
    percent: Percent,
    within: ThumbWithin ← all,
    type: Type,
    action: ThumbAction];
```

The **ThumbFeedbackProc** is provided to allow the client an opportunity to display his own feedback to the user concerning the relative location of the thumbing action DURING thumbing. Traditionally this feedback has been in the form of page numbers or a sideways arrow in the cursor, and/or information in the scrollbar itself (such as the Star diamond). **ThumbFeedbackProc** is first called when thumbing is invoked (user points down in the thumbing region of the scrollbar) with **action = down**. If the user moves the cursor the proc will be called again with **action = track**. When the user buttons up the proc will be called with **action. = up**. Clients should be aware that it is possible to never get the last call. The user may abort the thumb activity by either performing the pointUp outside of the scrollbar or by pressing the STOP key. The scrollbar code will be prepared to reset the Cursor but no other clean up will be provided.

```
ThumbScrollProc: TYPE = PROC[
    window: Window.Handle,
    percent: Percent,
    within: ThumbWithin ← all,
    type: Type];
```

The **ThumbScrollProc** is the procedure that actually performs the scroll. It is called when the user completes the thumb activity by pointing up in the thumbing region. The procedure is expected to make the appropriate calls to Window.SlideAndSize to cause the scroll to happen.

```
GetScrollProcs: PROCEDURE [window: Window.Handle, type: Type]
    RETURNS [
    single: SingleScrollProc,
    jump:JumpScrollProc,
    scrollbarInfo: ScrollbarInfoProc,
    thumb: ThumbScrollProc],
    feedback: ThumbFeedbackProc];
```

Returns the client scrollProcs associated with the viewer **window** and scrollbar **type**.

```
SetSingleScrollProc: PROCEDURE [
    window: Window.Handle, type: Type, scroll: SingleScrollProc] .
    RETURNS [old: SingleScrollProc];

SetJumpScrollProc: PROCEDURE [
    window: Window.Handle, type: Type, scroll: JumpScrollProc]
    RETURNS [old: JumpScrollProc];

SetScrollbarInfoProc: PROCEDURE [
    window: Window.Handle, type: Type, scroll: ScrollbarInfoProc]
    RETURNS [old: ScrollbarInfoProc];

SetThumbScrollProc: PROCEDURE [
    window: Window.Handle, type: Type, scroll: ThumbScrollProc]
    RETURNS [old: ThumbScrollProc];

SetThumbFeedbackProc: PROCEDURE [
    window: Window.Handle, type: Type, scroll: ThumbFeedbackProc]
    RETURNS [old: ThumbFeedbackProc];
```

Sets the various scrollProcs for **window**.

43.2.3 Utilities

```
GetScrollbar: PROCEDURE [window:Window.Handle, type: Type]
    RETURNS [scrollbar: Window.Handle];
```

Returns the **scrollbar** of **type** associated with **window**. Returns NIL if there isn't one.

```
GetScrollbarThickness: PROCEDURE RETURNS [INTEGER];
```

Returns the thickness of a scrollbar, i.e. the width of a vertical scrollbar or the height of a horizontal scrollbar. Useful for determining the overall size requirements for subwindows or shells.

```
GetZone: PROCEDURE [window:Window.Handle]
    RETURNS [zone: UNCOUNTED ZONE];
```

Returns the **zone** associated with **window** where **window** is the window the scrollbars are attached to.

PercentOf: PROCEDURE [v: INTEGER, p: Percent]
    RETURNS [INTEGER];

expresses **p** in terms of **v**
    example: m ← PercentOf[OutOfN, offset]

Percentage: PROCEDURE [part, full: INTEGER]
    RETURNS [Percent];

returns the percentage of **part** to **full**
    example: offset ← Percentage[m, OutOfN]

43.2.4 Errors

Error: ERROR [code: ErrorCode];

ErrorCode: TYPE = {alreadyExists, doesNotExist};

43.4 Index of Interface Items

44

Selection

44.1 Overview

The **Selection** interface defines the abstraction that is the user's current selection. It provides a procedural interface to the abstraction that allows it to be set, saved, cleared, and so forth. It also provides procedures that enable someone other than the originator of the selection to request information relating to the selection and to negotiate for a copy of the selection in a particular format.

44.1.1 Requestors and Managers

The **Selection** interface is used by two different classes of clients. Most clients wish merely to obtain the *value* of the current selection in some particular format; such clients are called *requestors*. These programs call **Convert** (or **ConvertNumber**, which in turn calls **Convert**), **Query**, or **Enumerate**. These clients need not be concerned with many of the details of the **Selection** interface.

The other class of clients consists of those who wish to own or set the current selection; these clients are called *managers*. A manager calls **Selection.Set** and provides procedures that may be called to convert the selection or to perform various actions on it. The manager remains in control of the current selection until some other program calls **Selection.Set**. These clients *do* need to understand most of the details of the **Selection** interface.

The goal of the **Selection** interface is that the requestor need never know, and should never care, what module is managing the selection. All that matters is whether the selection can be rendered in a suitable form. For example, suppose the user presses COPY and selects a printer icon as the destination. The printer implementation needn't know what is printable and what isn't. It simply queries the selection to determine whether it can be rendered as an Interpress master, and if so it obtains it and sends it. Otherwise, it queries whether the selection can be enumerated as a sequence of Interpress masters (as would be true of a folder, for instance). If this also fails, the object is rejected.

The selection is the expression of the user indicating the datum to be operated on. As such, it is conceptually owned by the user. The selection manager is a slave following the user's instructions.

To maintain this user interface model, the selection must only be changed at the explicit direction of the user. Software must allow the user to change the selection at will. To implement this user model, the selection is only changed from within the *user process* or *notifier*. The notifier is the system process that passes the user's actions, encoded as **TIP** results, to application software.

Software that wishes to read the selection must deal with the fact that the selection may be changed at any time that the notifier process is running. The way to synchronize with this potentially asynchronous activity is to only read the selection in the notifier process. This guarantees that the selection will not be altered while it is being read. Application software running in the notifier process can be assured that the selection will not change until after the application returns to the system. Thus the first rule for dealing with the selection is:

The selection may only be read or changed in the notifier process.

Once an application returns to the notifier, any knowledge it retains about the selection may be invalidated at any instant when the user subsequently changes the selection. Similarly, if an application running in the notifier passes some information about the selection to another process, that information may similarly be invalidated at any time. In these circumstances, the application must copy the selection's value, using **Copy**, **Move**, or **CopyMove**, to assure that its data remains valid. Thus the second rule for dealing with the selection is:

Copy the selection's value before returning to the system or before passing it to another process.

Copying the value of selection may not always be desirable. For example, if a selection requestor would like to copy a selection of files in the background, the requestor would not want to do a **Copy** before leaving the notifier; doing so would defeat the purpose of doing the filing operations asynchronously. To deal with the problem, **Selection** supports the notion of an *encapsulated* selection. While in the notifier, the requestor may ask that the selection be encapsulated. Doing so clears the user's selection, and saves the selection in a form that the selection manager can operate on from a non-notifier process. The requester can then call standard Selection requestor procedures passing in the encapsulated selection.

44.1.2 Essentials for a Requestor

Clients that need the *value* of the current selection.

44.1.2.1 Convert, Target, Value, Enumerate, CanYouConvert

The fundamental operation performed by a selection requestor is to obtain the value of the current selection by calling **Selection.Convert**. **Convert** takes a **Selection.Target** and returns a **Selection.Value**. The **Target** specifies what **TYPE** of data the selection should be converted to. The **Value** contains a pointer to the converted selection. For example, **Selection.Convert [target: string]** will return a pointer to a string, i.e., an **XString.Reader**.

Not all selections can be converted to all **Targets**; in fact most selections can be converted to only a small number of **Targets**. For example, if the selection is a text string, it can be converted to Target string and perhaps to integer, but probably not to file or fileType. Note: Converting to some Targets is not so much requesting the value of the selection as requesting some general information about the selection or its environment. For example, Selection.Convert [target: window] is a request for the window that the selection is in, Selection.Convert [target: help] is a request for user help information about the selection, etc. Note that Target is an open-ended enumeration and that clients can create new Targets by using Selection.UniqueTarget. The TYPE associated with each Target is determined by system-wide convention. Several of these TYPE/Target conventions are defined below under the description of Target. Other TYPE/Target conventions are documented in §44.2.1.1, Convert.

A requestor can also enumerate the selection if it is more than a single item or if it can be split into smaller pieces. This is done by calling Selection.Enumerate.

Finally, a requestor can determine what Targets the selection can be converted to without actually doing the conversion by calling Selection.CanYouConvert, Selection.Query, or Selection.HowHard.

If a requestor would like to be able to operate on the selection from outside the notifier, it can call SelectionX.Encapsulate. This clears the user's selection. Some managers may often lock the object(s) in the selection so that the user can't operate on them while they are encapsualated. Once the requestor has an encapsulated selection, all of the standard operations are available: ConvertX, EnumerateX, CanYouConvertX, QueryX, and HowHardX (all presently in the SelectionX interface). These operations are identical to the standard operations (Convert, etc.) but take an additional parameter for the encapsulated selection. Typically a requestor would do the Encapsulate in the notifier process, then fork a process to operation on the encapsulated selection.

44.1.2.2 Resource Allocation/Deallocation Considerations

It is a strict rule that the Values produced by Selection.Convert and Selection.Enumerate describe objects *owned by the selection manager*. The requestor may examine the data referenced by the value field, but must not alter it. Furthermore, the requestor must free the Value (using Selection.Free) once he no longer needs it.

If the requestor wishes to (1) keep the value after it returns to the system, or (2) pass the value to another process it must call Selection.Copy, Selection.Move, or Selection.CopyMove. These in turn invoke a procedure supplied by the selection manager that modifies the Value such that the requestor may then make changes to value↑ without affecting the selection manager. Fine Point: The procedure supplied by the manager is returned by the manager as part of the Value record. If a Move is performed, the item is also deleted from the manager's domain. After the Move or Copy, any storage associated with the Value is now owned by the requestor. This storage may be freed by calling Selection.Free.

For example, if the current selection is a document icon, then Convert[file] yields a Value containing a LONG POINTER TO NSFile.Reference for the file containing the document. If the requestor were to create a new document and associate it with the same file, it would probably have undesirable effects. Instead, the requestor should call Copy, passing in data:LONG POINTER TO NSFile.Reference for the destination directory of the new file. When Copy returns, the Value contains a reference to a copy of the original file, and the requestor can use this freely.

As a second example, suppose the selection manager uses a Mesa STRING as the internal selection representation. Then **Convert[string]** simply builds the string pointer into an **XString.Reader** using **XString.FromSTRING**. If the requestor wants to save the string for very long, he should call **Copy**, and the manager will allocate a copy of the original string using the **zone** passed to **Convert**. An alternative, somewhat simpler, is for the requestor to call **XString.CopyReader** or **XString.CopyToNewReaderBody** or **XString.CopyToNewWriterBody** to copy the bytes, and then call **Selection.Free** to dispose of the original **Reader**.

An alternative to copying the selection before using it in another process is encapsulating the selection. See the discussion in §44.2.2.7.

44.1.3 Essentials for a Manager

Clients that own and manage the current selection.

44.1.3.1 Set, ConvertProc, ActOnProc, ManagerData

The implementor of a selection manager needs to know everything that the implementor of a selection requestor knows, plus more (see the previous section **Essentials for a Requestor**).

The fundamental operation performed by a selection manager is to become the current manager by calling **Selection.Set**. **Set** takes a **ConvertProc**, an **ActOnProc**, and a LONG POINTER (**ManagerData**).

The **ConvertProc** is called to obtain the value of the selection, whenever a requestor calls **Selection.Convert** or **Selection.Enumerate**. The **ConvertProc** is also called to determine what **Targets** the selection can be converted to, whenever a requestor calls **Selection.CanYouConvert**, **Selection.Query**, or **Selection.HowHard**. **ConversionInfo** is a variant record passed to the **ConvertProc** that indicates which operation to perform: **convert**, **enumeration**, or **query**.

The **ActOnProc** is called to perform various **Actions** on the selection, such as **mark**, **unmark**, **clear**, and **encapsulate**.

The **ManagerData** passed to **Set** is passed back to the **ConvertProc** and the **ActOnProc**. Typically, the **ManagerData** identifies exactly what portion of the manager's domain is currently selected. For example, if the current selection is some text in a document, the actual manager is the document application, which has some **ManagerData** that indicates exactly which characters are currently selected.

When a manager calls **Selection.Set**, the previous manager is told to **ActOn [clear]**, and **Selection** forgets about the previous manager. Hence, there is only one selection at a time. However, **Selection** also supports the notion of a "saved" selection. A client can become the current manager by calling **Selection.SaveAndSet**, which does a **Set** but also saves the previous selection. Later, the manager that did the **SaveAndSet** can do a **Selection.Restore**, which restores the previous selection.

Related to the notion of a saved selection is an encapsulated selection. The difference between the two is that a saved selection cannot be operated on until it is restored. An encapsulated selection can be operated on using the **ConvertX**, **EnumerateX**, etc. operations that all take a parameter for the encapsulated selection. To encapsulate the selection, the requestor calls SelectionX.Encapsulate which calls the managers's ActOnProc with action unmark followed by ActOn[encapsulate]. The manager is expected to modify its manager data that was passed into Set to note that the selection is saved. Th manager should also lock down the objects in the selection if necessary or do whatever is necessary to insure that the objects in the selection will remain accessible until the encapsulated selection is freed. When the requestor later operates on the encapsulated selection, the manager's ActOnProc and ConvertProc will be called with the manager data that was originally passed into Selection.Set.

44.1.3.2 More on Selection.Value, ValueFreeProc, and ValueCopyMoveProc

The Value produced by a manager's ConvertProc contains more than simply a pointer to the converted selection. It also contains a pointer to two procedures, a ValueFreeProc and a ValueCopyMoveProc. The ValueFreeProc is called when the requestor calls Selection.Free so that the manager can release any resources that were allocated when the selection was converted. The manager's ValueCopyMoveProc is called when the requestor calls Copy, Move, or CopyMove. The ValueCopyMoveProc should copy or move the converted selection value so that the manager no longer owns the resources associated with the value. A third field in the Value record is a LONG UNSPECIFIED that may be used to store data for the ValueFreeProc and the ValueCopyMoveProc.

If the converted selection value can be copied or moved, the manager must return a ValueCopyMoveProc with the Value. For example, Targets string and file can be moved or copied, while it does not make sense to move or copy Targets window and fileType. The ValueCopyMoveProc modifies the Value such that the requestor may then make changes to value↑ without affecting the selection manager. If a Move is performed, the item is also deleted from the manager's domain. (Some managers may implement Copy but raise Error[invalidOperation] if asked to do a Move.) The interpretation of the data given to a ValueCopyMoveProc depends on the manager; the typical use is to specify a destination for the object.

44.1.3.3 Storage Considerations for ConvertProc

As stated above, it is a strict rule that the Values produced by the ConvertProc describe objects *owned by the manager*. If the manager allocated any resources to produce the converted selection value, then a ValueFreeProc must be returned with the Value so that the resources can be released. If a ValueCopyMoveProc was returned with the Value, after the converted selection value has been copied or moved, the manager must ensure that the correct things will happen when the Value's ValueFreeProc is called (i.e., when the requestor calls Selection.Free). This may involve replacing the original ValueFreeProc.

The manager's ConvertProc takes a zone that Selection guarantees is valid (except for the query operation). The manager should allocate any storage for the converted selection value from that zone. The ConvertProc can store the zone in the context (LONG UNSPECIFIED) field of the Value record (or in a record pointed to by the context field). The ValueFreeProc and ValueCopyMoveProc can then retrieve this zone to free the storage.

Numerous defaults are provided by Selection to ease the manager's task of proper storage management. In practice, the ConvertProc can simply default the context field and Selection will place the zone there. Also, procedures such as FreeStd and FreeContext are provided that perform the LOOPHOLEs, FREE the storage, and store null and/or no-op values such as **NopFree** in the **Value** record.

44.1.3.4 Storage Considerations for ManagerData

The **ManagerData** that identifies exactly what part of the manager's domain is currently selected should be allocated whenever a **Selection.Set** is done and deallocated whenever **ActOn [clear]** is requested. In particular, a manager should not assume that there will be only one selection at a time in his domain. The existence of **SaveAndSet** and **Restore** and **Encapsulate** implies that the same manager *code* could have several pushed selections at once and therefore would have several **ManagerData** records allocated at once.

A manager that supports encapsulated or saved selections must carefully manage storage and access to the objects represented by the encapsulated selection. The saved selection case is slightly easier; the manager can assume that the **ManagerData** will not be operated on unless it is the user's selection. When the saved selection is restored, the manager must make sure all the objects in the saved selection still exist. The encasulated case is somewhat harder. One approach is to save enough data about the selection to work properly if some of the objects disappear. For example, a folder application may normally maintain a selection as a span of objects: rows 1-5 are selected. To encapsulate the selection, the folder might have convert the selection into a list of fileIds so it can later access each file. Another approach (and the one used currently in the BWS) is to lock all of the objects in the encapsulated selection so that that user cannot modify them unto the requestor calls **Selection.Discard**.

44.2 Interface Items

44.2.1 Requestor items

44.2.1.1 Convert

```
Convert: PROCEDURE [target: Target, zone: UNCOUNTED ZONE ← NIL]
  RETURNS [value: Value];

ConvertX: PROCEDURE [
  target:Target,
  zone: UNCOUNTED ZONE ← NIL,
  manager: Saved ← nullManager]
  RETURNS [value:Value];

Value: TYPE = RECORD [value: LONG POINTER, . . .];

nullValue: Value = [value: NIL, . . .];
```

Convert is a request to the current selection manager to produce the selection as a TYPE specified by **target**, if possible. **value.value** will be a LONG POINTER TO the converted selection. The TYPE of object pointed to by **value.value** depends on **target** and is described below under **Target**. If the conversion requires that storage be allocated, it will be allocated out of **zone**. If **zone** is defaulted, the system heap is used.

The value returned is read-only; it belongs to the manager. If the requestor wishes to (1) keep the value after it returns to the system, or (2) pass the value to another process, it must call **Copy**, **Move**, or **CopyMove** to make a copy of the value, which is then owned by the requestor. If **Copy**, **Move**, or **CopyMove** is called, the requestor must still call **Free**.

If **Copy**, **Move**, or **CopyMove** is not called, the requestor must call **Free** after calling **Convert**. This allows the manager to free any resources that were allocated to perform the conversion. If **Copy**, **Move**, or **CopyMove** is called, the requestor then owns any resources and may retain them indefinitely and/or may free them by calling **Free**.

There are other fields in the **Value** record, but the requestor need not be concerned with them. They are described in the section on **Manager Items**

nullValue is returned if the selection manager does not implement the desired conversion, or if the particular selection is incompatible with the target (e.g., **Convert[integer]** when non-numeric characters are selected).

ConvertX adds an addition **manager** parameter to support encapsulated selections. manager is obtained by doing a **SelectionX.Encapsulate**. If **manager** is defaulted, the operation applies the the global user's selection; otherwise the manager is called to convert the encapsulated selection represented by **manager**. **ConvertX** is exported by **SelectionX**.

```
Target: TYPE = MACHINE DEPENDENT{
   window(0), shell, subwindow, string, length, position,
   integer, interpressMaster, file, fileType, token, help,
   interscriptScript, interscriptFragment, serializedFile, name, firstFree, last(1777B)};

fileWithFeedback:Target;
```

Target describes the type of data to which a selection may be converted (see **Convert**). Modules that manage the current selection may choose not to implement conversion to some (or even most) of these types. The values described below are those stored in the **value** field of the **Selection.Value** returned by **Convert**. **fileWithFeedback** is a target presently exported by **SelectionX**.

Special note for **Targets** that produce a **Stream.Handle**: The **Stream.Object** pointed to by the **Stream.Handle** is read-only. Thus the requestor cannot even read the stream because that alters the stream state and thus the **Stream.Object**. Before using the stream, the requestor must do a **Copy**, after which the ownership of the storage for the **Stream.Object** and any of its ancillary data moves to the requestor. Note also that the stream itself is read-only even after the **Copy**. The requestor should never attempt to write to the stream. After reading the stream, the requestor can free the stream and any associated resources by calling **Stream.Delete**. Thus a typical stream requestor will do **Convert[stream]; Copy[]; <read stream>; Stream.Delete[];** Note for selection managers: this last point means that the **Stream.Delete** must be able to free any ancillary data associated with the stream.

Note that some **Target** values refer to types that are not defined within the context of ViewPoint. Such targets so far include **pieceList**, **help**, **interscriptScript**, and **interscriptFragment**. Popular target types are included in the **Selection** interface as a convenience for clients. New target types will be put either into a **SelectionExtras** interface or, for little-used types, into private interfaces negotiated between managers and requestors and using **Selection.UniqueTarget**. The **TYPE** associated with each **Target** is determined by system-wide convention. Several of these **TYPE/Target** conventions are defined here. Other TYPE/Target conventions are documented elsewhere, see §44.2.2.8, **UniqueTarget**.

Fine Point: This **Selection** interface is intended to support both XDE and ViewPoint clients, so there may be **Targets** that do not make sense in one domain or the other. **Targets** that only make sense in one domain show that domain in parentheses.

window	yields a **Window.Handle** for the window containing the selection.
shell	yields a **StarWindowShell.Handle** for the window containing the selection. (ViewPoint)
subwindow	yields a **Window.Handle** for the subwindow containing the selection. (XDE)
string	yields a LONG POINTER TO **XString.ReaderBody** (an **XString.Reader**) representing the text of the selection. If the current selection is too large, the manager of the selection may return **nullValue** when asked to convert to a **string**. The requestor should then ask to enumerate the selection as a sequence of smaller **strings**. **Note:** The requestor must copy the **ReaderBody** before altering it.
length	yields a LONG POINTER TO LONG CARDINAL containing the length of the selection in characters.
position	yields a LONG POINTER TO LONG CARDINAL containing the position within the source.
pieceList	yields a list of pieces, understood by the internals of XDE's **PieceSource** interface. (XDE)
integer	yields a LONG POINTER TO LONG INTEGER containing the result of converting the contents of the selection to a number.
interpressMaster	yields a **Stream.Handle** onto an Interpress master, according to the Interpress standard.
file.	yields a LONG POINTER TO **NSFile.Reference** for the file (if any) associated with the selection, e.g., the backing file for a Star document icon. When calling **Copy**, **Move**, or **CopyMove**, the **data** parameter must be a LONG POINTER TO **NSFile.Reference** of the parent directory to where the file should be copied or moved. After calling **Copy**, the **value.value** is replaced by a LONG POINTER TO **NSFile.Reference** of the newly copied file. (ViewPoint)
fileWithFeedback	same as **file** but tells the manager's **CopyMoveProc** to post the names of the objects to the attention window as they are copied or moved. (ViewPoint)

fileType	yields a LONG POINTER TO NSFile.Type for the file (if any) associated with the selection. (ViewPoint)
token	yields a LONG POINTER TO XString.ReaderBody (an XString.Reader) that contains the first token of the current selection. **Note:** The requestor must copy the ReaderBody before altering it.
help	yields a LONG POINTER to a value or data structure that specifies what should happen if the HELP key is pressed. Consult the Help documentation (not in this manual) for the exact TYPE.
interscriptScript	yields a Stream.Handle onto a complete script, according to the Interscript standard. It begins with the "Interscript 1.0 . . .", and is in machine code.
interscriptFragment	yields a Stream.Handle onto a single Interscript node, in machine code.
serializedFile	A Target of serializedFile results in a Stream.Handle. Stream.GetXXX operations can be performed on the stream. This is useful for retrieving files from non-NSFile mediums such as a floppy disk.
name	A Target of name results in a XString.Reader that contains the name of the object.
firstFree	is used internally by UniqueTarget and should not be used by clients.

ConvertNumber: PROCEDURE [target: Target]
  RETURNS [ok: BOOLEAN, number: LONG UNSPECIFIED];

This procedure lets the requestor streamline his code in many cases. ConvertNumber calls Convert and assumes that the resulting value.value references a 32-bit object. (This is true of the targets length, position, integer, and fileType, and may also be true of targets defined using UniqueTarget.) The object is returned as number, and the Value is then freed (Selection.Free). If the selection manager does not support the desired conversion (that is, it returns nullValue), or if the selection could not be converted to a number, ConvertNumber returns ok:FALSE; otherwise, it returns ok:TRUE.

Free: PROCEDURE [v: ValueHandle];

ValueHandle: TYPE = LONG POINTER TO Value;

Free allows the manager to free any storage associated with v. Free should always be called after calling Convert (but if Copy or Move is going to be called, don't call Free until after calling Copy or Move). Fine point: after calling Copy or Move, Free is a no-op since the manager should have handed over any storage ownership to the requestor, but it is easier for the requestor to simply remember the rule, "Always call Free." The manager will take care of ensuring that an extraneous Free is harmless.

44.2.1.2 Query

A requestor can determine exactly which **Targets** the current selection can be converted to and how difficult the conversion would be. The most common way to do this is **CanYouConvert**, which takes a **Target** and returns a BOOLEAN indicating whether the selection can be converted to that **Target**. **HowHard** is similar to **CanYouConvert** but returns a **Difficulty**. **Query** allows a requestor to determine the **Difficulty** of conversion for an ARRAY of **Targets**.

Note: For all these queries, the manager is indicating how hard it would be *to attempt* to convert the selection to that target type. Attempt is a key word. The manager might be willing to attempt to convert the selection to an Interpress master and yet run out of disk space when the conversion is actually requested. Likewise, the manager might support conversion to integer, but the conversion could still fail if the selection contains invalid characters.

```
CanYouConvert: PROCEDURE [target: Target, enumeration: BOOLEAN ← FALSE]
  RETURNS [yes: BOOLEAN] = INLINE {
    RETURN [ HowHard [ target, enumeration ] # impossible ] };

CanYouConvertX: PROCEDURE [
  target:Target,
  enumeration: BOOLEAN ← FALSE,
  manager: Saved ← nullManager]
  RETURNS [yes: BOOLEAN] = INLINE{
    RETURN[ HowHardX[ target, enumeration, manager ] # impossible] };
```

CanYouConvert determines whether the selection manager supports conversions to the specified **target**. **enumeration** = TRUE means the requestor wants to know if the manager supports enumerating the selection in the specified target form. (See the section on Enumeration below.) **CanYouConvertX** is the same procedure but can operate on an encapsulated selection **manager**. If **manager** is defaulted, the operation uses the default global selection. **CanYouConvertX** is exported by **SelectionX**.

```
HowHard: PROCEDURE [target: Target, enumeration: BOOLEAN ← FALSE]
  RETURNS [difficulty: Difficulty];

HowHardX: PROCEDURE [
  target:Target,
  enumeration: BOOLEAN ← FALSE,
  manager: Saved ← nullManager]
  RETURNS [difficulty:Difficulty];

Difficulty: TYPE = {easy, moderate, hard, impossible};
```

HowHard determines the **difficulty** the selection manager would have attempting to convert to the specified **target**. **enumeration** = TRUE means the requestor wants to know the **difficulty** of enumerating the selection in the specified target form. (See the section on Enumeration below.) **HowHardX** is the same procedure but can operate on an encapsulated selection **manager**. If **manager** is defaulted, the operation uses the default global selection. **HowHardX** is exported by **SelectionX**.

The difficulty ratings are interpreted roughly as follows:

easy	Requires virtually no computation (other than allocating storage for the **Value**). Example: **length** when the selection is being maintained as two character indices within a string.
moderate	Requires some amount of computation but nothing outrageously time-consuming. Example: converting the above-mentioned substring representation to a **string** or **integer** target.
hard	Requires extensive computation. Example: **interpressMaster**.
impossible	The selection manager does not support this conversion.

```
Query: PROCEDURE [targets: LONG DESCRIPTOR FOR ARRAY OF QueryElement];

QueryX: PROCEDURE [
   targets: LONG DESCRIPTOR FOR ARRAY OF QueryElement,
   manager: Saved ← nullManager];

QueryElement: TYPE = RECORD [
   target: Target,
   enumeration: BOOLEAN ← FALSE,
   difficulty: Difficulty ← TRASH];
```

Query allows a requestor to determine the difficulty of conversion for several **Targets**. The requestor should construct the ARRAY OF **QueryElement**, filling in **target** and **enumeration** for each **QueryElement**. The manager will then store a **Difficulty** in each **QueryElement** indicating how hard it would be to attempt to convert the selection to that **target**. The requestor can then examine the **difficulty** field of each **QueryElement** after the call to **Query**. **QueryX** is the same procedure but can operate on an encapsulated selection **manager**. If **manager** is defaulted, the operation uses the default global selection. **QueryX** is exported by **SelectionX**.

44.2.1.3 Enumeration

The selection is sometimes a collection of items (for example, several rows of a folder) or a single large item that can be split up (for example, a long string can be broken into several smaller ones). A requestor can request that each item or part of such selections be converted to some **Target** by calling **Selection.Enumerate**. **Enumerate** is logically similar to calling **Convert** for each item and the same storage ownership rules apply (see **Convert**). Not all selection managers support enumerating the selection; for example, they do not support a selection that is more than one item. Often a requestor will call **Convert** and if that fails (returns **nullValue**), call **Enumerate**.

```
Enumerate: PROCEDURE [
   proc: EnumerationProc, target: Target, data: RequestorData ← NIL,
   zone: UNCOUNTED ZONE ← NIL]
   RETURNS [aborted: BOOLEAN];
```

```
EnumerateX: PROCEDURE [
  proc: EnumerationProc, target: Target, data: RequestorData ← NIL,
  zone: UNCOUNTED ZONE ← NIL,
  manager: Saved ← nullManager]
  RETURNS [aborted: BOOLEAN];
```

```
EnumerationProc: TYPE = PROCEDURE [element: Value, data: RequestorData]
   RETURNS [stop: BOOLEAN ← FALSE];
```

```
RequestorData: TYPE = LONG POINTER;
```

Enumerate is a request to the selection manager to enumerate the current selection, converting each **element** to **target**. **proc** is called for-each **element**. **data** is passed back to **proc** each time it is called. As with the **Value** returned by **Convert**, *the requestor must consider each* **element** *to be read-only* until **Copy**, **Move**, or **CopyMove** is called, and the requestor must free the value by calling **Free** for each **element**. **Free allows the manager to** free any storage associated with **element**. **Free** should always be called for each element (but if **Copy** or **Move** is going to be called, don't call **Free** until after calling **Copy** or **Move**). Fine point: after calling **Copy** or **Move**, **Free** is a no-op since the manager should have handed over any storage ownership to the requestor, but it is easier for the requestor to simply remember the rule, "Always call Free." The manager will take care of ensuring that an extraneous Free is harmless.

stop is returned from **proc** by the requestor and indicates whether the enumeration should be stopped. **aborted** indicates whether the enumeration completed normally or terminated prematurely.

If the manager cannot convert the selection to the target type or if the manager does not implement enumeration, **proc** will not be called.

EnumerateX is the same procedure but can operate on an encapsulated selection **manager**. If **manager** is defaulted, the operation uses the default global selection. **EnumerateX** is exported by **SelectionX**.

Warning: the requestor must not do anything inside of **proc** that would cause **Selection** to be called (**Clear**, for example) since this will result in a monitor lock.

```
Reconversion: SIGNAL [
   target: Target, zone: UNCOUNTED ZONE] RETURNS [Value];
```

```
ReconvertDuringEnumerate: PROCEDURE [
   target: Target, zone: UNCOUNTED ZONE ← NIL] RETURNS [Value];
```

A requestor may wish to reconvert the current item during an enumeration of the selection. The requestor should call **ReconvertDuringEnumerate**, which will raise the signal **Reconversion**. If the manager supports reconversion, it should catch the signal and return the reconverted value. If the manager does not support reconversion, it should ignore the signal. **Enumerate** will catch the signal and return **nullValue**. **ReconvertDuringEnumerate** acts like **Convert** with respect to **zone**.

```
maxStringLength: CARDINAL = ...;
```

maxStringLength is obsolete.

44.2.1.4 Copy, Move, Free, etc.

The **Values** produced by **Convert** and **Enumerate** are *strictly read-only. The storage is owned by the manager.* The requestor may examine the data referenced by the **value** field but must not alter it.

If the requestor wishes to (1) keep the value past when it returns to the system, or (2) pass the value to another process, it must call **Copy**, **Move**, or **CopyMove**. These in turn invoke a procedure supplied by the selection manager that modifies the **Value** such that the requestor may then make changes to **value.value↑** without affecting the selection manager. Fine Point: This procedure is returned by the manager as part of the **Value** record, but the requestor never needs to know about these details. If a **Move** is performed, the item is also deleted from the manager's domain. After the **Move** or **Copy**, any storage associated with the **Value** is now owned by the requestor. This storage may be freed by calling **Free**.

For example, if the current selection is a document icon, then **Convert[file]** yields a **Value** containing a LONG POINTER TO **NSFile.Reference** for the file containing the document. If the requestor were to create a new document and associate it with the same file, it would probably have undesirable effects. Instead, the requestor should call **Copy**, giving it a LONG POINTER TO **NSFile.Reference** for the destination directory of the new file. When **Copy** returns, the **Value** contains a reference to a copy of the original file, and the requestor can use this freely. Furthermore, whereas calling **Free** with the original **Value** might have deleted the file (since the file then belonged to the manager, who might have created it solely for the **Convert** request), calling **Free** for the new **Value** frees only the **NSFile.Reference** storage (since the file is now a permanent object belonging to the requestor).

```
Copy: PROCEDURE [v: ValueHandle, data: LONG POINTER] = INLINE {
  CopyMove[v, copy, data]};

Move: PROCEDURE [v: ValueHandle, data: LONG POINTER] = INLINE {
  CopyMove[v, move, data]};

CopyMove: ValueCopyMoveProc;

ValueCopyMoveProc: TYPE = PROCEDURE [
  v: ValueHandle, op: CopyOrMove, data: LONG POINTER];

CopyOrMove: TYPE = {copy, move};
```

Copy, **Move**, and **CopyMove** request the manager to make a copy of the converted selection value (**v.value↑**) and, for **Move**, also delete the selection from the manager's domain. A requestor may call these procedures after calling **Convert** or from an **EnumerationProc** while doing an **Enumerate**. **data** will be passed to the manager; what it points to depends on the particular **Target**. **data** often points to a destination container for the copied value. For example, for **Target file**, **data** is a LONG POINTER TO **NSFile.Reference** for the destination directory. The exact meaning of **data** for each target is specified in the description of that target under **Target** above. **Copy**, **Move**, and **CopyMove** may raise **Error [invalidOperation]**.

Encapsulated selections also work with **Copy** and **Move**: if a requestor has an encapsulated selection and obtains a value with **ConvertX** or **EnumerateX**, they can use all of the standard operations that work with a **Value**.

44.2.2 Manager Items

44.2.2.1 Set

```
Set: PROCEDURE [pointer: ManagerData, conversion: ConvertProc, actOn: ActOnProc];

ManagerData: TYPE = LONG POINTER;
```

The **Set** procedure allows a client to become the manager of the current selection by supplying the **Selection** interface with a pair of procedures. The **ActOnProc** is called to modify or manipulate the current selection. The **ConvertProc** is called to get the value of the current selection. The value of **pointer** passed to **Set** is used as the **data** argument in calls to **conversion** or **actOn**. **pointer** typically points to a record that describes what part of the manager's domain is currently selected. If there is already a selection manager when **Set** is called, **Set** first calls that manager with **ActOn[unmark]** and **ActOn[clear]**. **Set** automatically calls the new **ActOnProc** with an action of **mark**.

Either **conversion** or **actOn** can be explicitly NIL. If **conversion** is NIL, then **Convert** always returns **nullValue**, **Enumerate** is a no-op, and **Query** will always respond **impossible**. If **actOn** is NIL, then **ActOn** is a a no-op for all actions.

```
ConvertProc: TYPE = PROCEDURE [
   data: ManagerData,
   target: Target,
   zone: UNCOUNTED ZONE,
   info: ConversionInfo ← [convert[]] ]
   RETURNS [value: Value];

ConversionInfo: TYPE = RECORD [SELECT type: * FROM
   convert = > NULL,
   enumeration = > [proc: PROCEDURE [Value] RETURNS [stop: BOOLEAN]],
   query = > [query: LONG DESCRIPTOR FOR ARRAY OF QueryElement],
   ENDCASE];
```

A **ConvertProc** is provided by a manager when becoming the manager; that is, when calling **Set** or **SaveAndSet**. The manager's **ConvertProc** is called when a requestor calls **Convert**, **Enumerate**, or **Query**. The **ConvertProc** should perform the conversion, the enumeration, or the query. **info** is a variant record indicating which operation to perform: it contains data appropriate to each operation. The **ConvertProc** should use WITH **info** SELECT to determine which operation is requested. Each operation is described in detail in the following sections. **data** is the pointer that was passed to **Set** or **SaveAndSet** and typically points to a record that describes what part of the manager's domain is currently selected. **target** indicates the TYPE of object that the selection should be converted to and is meaningful only for conversion and enumeration. **zone** should be used to allocate any storage for the converted selection value and is meaningful only for conversion and enumeration.

```
ActOnProc: TYPE = PROCEDURE [data: ManagerData, action: Action]
  RETURNS [cleared: BOOLEAN ← FALSE];
```

An **ActOnProc** is provided by the manager of the selection to perform various actions on the selection. **ActOnProc** is fully described later in this chapter.

44.2.2.2 Conversion

```
ConversionInfo: TYPE = RECORD [SELECT type: * FROM
  convert = > NULL,
  . . .
  ENDCASE];

Value: TYPE = RECORD [
  value: LONG POINTER,
  ops: LONG POINTER TO ValueProcs ← NIL,
  context: LONG UNSPECIFIED ← 0];
```

Convert calls the manager's **ConvertProc** with **convert ConversionInfo** to perform the requested conversion. The **ConvertProc** returns a **value: Value**. If the conversion can be performed, **value.value** should point to the converted selection value; **value.ops** should point to a pair of procedures, a **ValueFreeProc** that will release any resources that were allocated to perform the conversion and a **ValueCopyMoveProc** that will copy or move the converted value; **value.context** can be used to save any information that the pair of procedures might need. **value.ops** and **value.context** are described in much more detail later. If the manager does not support the requested **Target** or there is some problem with the conversion, the **ConvertProc** should return **nullValue**. See **Target** for the effect of different conversion targets.

If the conversion requires that an object be allocated, the **ConvertProc** should allocate it out of **zone**. If the requestor passed a NIL zone to **Convert**, **Convert** passes the system zone to **ConvertProc**. The **ConvertProc** can assume that it is always given a valid **zone**.

44.2.2.3 Query

```
ConversionInfo: TYPE = RECORD [SELECT type: * FROM
  . . .,
  query = > [query: LONG DESCRIPTOR FOR ARRAY OF QueryElement],
  ENDCASE];

QueryElement: TYPE = RECORD [
  target: Target,
  enumeration: BOOLEAN ← FALSE,
  difficulty: Difficulty ← TRASH];
```

Query, **HowHard**, and **CanYouConvert** call the manager's **ConvertProc** with **query ConversionInfo**. The **ConvertProc** should examine the **target** and **enumeration** fields of each **QueryElement** (these were filled in by the requestor) and fill in the **difficulty** field indicating how hard it would be to attempt to convert the selection to that **target** (**enumeration = FALSE**) or to convert the selection to an enumeration of that target (**enumeration = TRUE**).

All managers are expected to implement queries; the assumption is that most difficulty ratings can be determined simply by indexing into a constant array. The **Value** actually returned by the **ConvertProc** in response to a query is ignored; **nullValue** or **TRASH** may be returned.

Note that the manager is indicating how hard it would be to attempt to convert the selection to that target type. Attempt is a key word. The manager might be willing to attempt to convert the selection to an Interpress master, and yet run out of disk space when the conversion is actually requested. Likewise, the manager might support conversion to integer, but the conversion could still fail if the selection contains invalid characters.

44.2.2.4 Enumeration

```
ConversionInfo: TYPE = RECORD [SELECT type: * FROM
   ...,
   enumeration = > [proc: PROCEDURE [Value] RETURNS [stop: BOOLEAN]],
   ...,
   ENDCASE];
```

Enumerate calls the manager's **ConvertProc** with **enumeration ConversionInfo**. The **ConvertProc** should convert each element or part of the selection according to **target** and call **proc** for each element. The **Value** passed to **proc** is just as it is for conversion (see the section on **Conversion** above and the following section). If **proc** returns **stop = TRUE**, the **ConvertProc** should stop the enumeration and return. The **Value** returned by the **ConvertProc** after an enumeration is ignored; **nullValue** or **TRASH** may be returned. Not all selection owners are expected to implement enumerations; if an enumeration is requested and not supported, the **ConvertProc** should simply return and take no other action. Fine Point: The **ConvertProc** does not call the requestor's **EnumerationProc** directly; rather, **proc** is inside **Enumerate** and **Enumerate** calls the requestor's **EnumerationProc**. This lets **Enumerate** insert the **zone** into the **Value.context** if it is zero, just as **Convert** does for **Values** produced by a simple conversion.

```
maxStringLength: CARDINAL = ...;
```

maxStringLength is obsolete.

44.2.2.5 Free, Copy, Move, etc.

```
ValueHandle: TYPE = LONG POINTER TO Value;
```

```
Value: TYPE = RECORD [
   value: LONG POINTER,
   ops: LONG POINTER TO ValueProcs ← NIL,
   context: LONG UNSPECIFIED ← 0];
```

The selection manager provides the value of the selection, or other selection-related information, to the requestor by means of **Value** records. These records are typically either returned by a **ConvertProc** or passed as elements to the requestor's **EnumerationProc**. The **ops** field defines the effect of **Free**, **Copy**, **Move**, and **CopyMove**. The **context** field may be used to store data for use by the **ops** procedures. If the **context** field is defaulted (zero) by the selection manager, **Selection** stores the **zone** that was passed to the **ConvertProc** there before the **Value** is handed to the requestor.

```
ValueProcs: TYPE = RECORD [
  free: ValueFreeProc ← NIL,
  copyMove: ValueCopyMoveProc ← NIL];
```

ValueProcs are returned by the manager as part of a **Value** record. If the manager allocated any resources to produce the converted selection value, then a **ValueFreeProc** must be returned with the **Value** so that the resources can be released. **free** is called when the requestor calls **Free**. If the converted selection value can be copied or moved, the manager must return a **ValueCopyMoveProc** with the **Value**. For example, **Targets string** and **file** can be moved or copied, while it does not make sense to move or copy **Targets window** and **fileType**. **copyMove** will be called when the requestor calls **Copy**, **Move**, or **CopyMove**.

44.2.2.5.1 Free

```
ValueFreeProc: TYPE = PROCEDURE [v: ValueHandle];
```

If any resources were allocated to produce the converted selection value, they should be released in the manager's **ValueFreeProc**. The **ValueFreeProc** is returned as part of the **ops** field of a **Value**. The **ValueFreeProc** will be called when the requestor calls **Free**. **v** points to the **Value** that represents the converted selection.

Defaults are provided such that for the most common case when the **ConvertProc** simply allocates one node of storage from the passed **zone**, the manager need not supply a **ValueFreeProc**. **Selection** takes care of freeing the storage when the requestor calls **Free**. The details of how this works are as follows:

The manager's **ConvertProc** takes a **zone** that **Selection** guarantees is valid. The manager should allocate any storage for the converted selection value from that **zone**. The **ConvertProc** can store the **zone** in the **context** field of the **Value** record (or in a record pointed to by the **context** field); then the **ValueFreeProc** can retrieve this **zone** to free the storage. **Selection** stores this **zone** in the **context** field if **context** is zero (the default) in the **Value** returned by the **ConvertProc** (or passed to the **EnumerationProc**). **v.value** points at the converted selection object to be freed. Now, **Free** calls **FreeStd** if the **Value** passed to **Free** has **ops = NIL** or **ops.free = NIL**. **FreeStd** treats **v.context** as a **ZONE** and calls **v.context.FREE[@v.value]**.

If there are in fact no resources that should be freed (for example, after **Convert[window]**), the selection manager should use **NopFree** as the **ValueFreeProc**. (See also **nopFreeValueProcs**.)

```
FreeStd: ValueFreeProc;
```

FreeStd assumes the resources of the **Value** can be freed by treating **v.context** as a **ZONE** and calling **v.context.FREE[@v.value]**. If a **Value** has **ops = NIL** or **ops.free = NIL**, **Free** will call **FreeStd**.

```
NopFree: ValueFreeProc;
```

The **NopFree** procedure should be used as the **ops.free** for a **Value** involving no temporary resources owned by the selection manager. Thus, a **Value** created by **Convert[window]** would probably use **NopFree**, as would **Convert[string]** if the **Value.value** pointed to a permanent **XString.ReaderBody** belonging to the manager.(See also **nopFreeValueProcs**.)

44.2.2.5.2 Copy and Move

```
ValueCopyMoveProc: TYPE = PROCEDURE [
  v: ValueHandle, op: CopyOrMove, data: LONG POINTER];
```

```
CopyOrMove: TYPE = {copy, move};
```

The manager's **ValueCopyMoveProc** is called to copy or move the converted selection value. A **ValueCopyMoveProc** is returned by the manager's **ConvertProc** as part of the **ops** field of a **Value**. The **ValueCopyMoveProc** is called when the requestor calls **Copy**, **Move**, or **CopyMove**. The **ValueCopyMoveProc** should modify the **Value** such that it no longer involves any manager-owned storage. If a **Move** is performed, the item is also deleted from the manager's domain. (Some managers may implement **Copy** but raise **Error[invalidOperation]** if asked to do a **Move**.) **data** is the **data** parameter that the requestor passed to copy or move. It is often a pointer to the destination container for the copied value. The interpretation of **data** depends on the **Target**; it is specified in the description of each target under **Target** above. **v** points to the **Value** representing the converted selection. **op** indicates whether to do a copy or move. **Note: v.context** can be used by the manager to save information between the **ConvertProc** and the **ValueCopyMoveProc**.

The **ValueCopyMoveProc** should release (or perhaps simply turn over control of) any resources that were allocated by the **ConvertProc** to produce the original converted value. Conceptually, the **ValueCopyMoveProc** makes a copy of the converted value, then releases any resources that were used to produce the original converted value. If the original converted value itself was a copy produced by the conversion process, this effect might be achieved by doing nothing -- the requestor just becomes the owner of the copy.

If the converted value can only be copied once (the typical case), the **ValueCopyMoveProc** should also set **v.ops.copyMove** to **NIL** to prevent the manager's **ValueCopyMoveProc** from being called again. If the requestor does call **Copy** or **Move** again, **Selection** raises **Error [invalidOperation]**.

The **ValueCopyMoveProc** should also ensure that **v.ops.free** and **v.context** have appropriate values so that when the requestor calls **Free**, the right thing happens. For example, if the newly copied selection was allocated from a zone, **v.ops.free** should free it from that zone (see **ValueFreeProc** and **FreeStd**); or if the newly copied selection has no storage allocated for it, **v.ops.free** should be **NopFree**.

```
nopFreeValueProcs: READONLY LONG POINTER TO ValueProcs; -- @[NopFree, NIL]
```

This is provided for use as the **ops** vector in **Values** that require no temporary storage and that cannot be moved or copied. The **window** and **subwindow Targets** typically produce such values.

```
FreeContext: PROCEDURE [v: ValueHandle, zone: UNCOUNTED ZONE] = INLINE {
  zone.FREE[LOOPHOLE[@v.context, LONG POINTER TO LONG POINTER]];
  v.context ← LOOPHOLE[zone]};
```

When the requestor calls **Copy** or **Move**, the manager's **ValueCopyMoveProc** is expected to modify the **Value** that it no longer involves any manager-owned storage. If the manager has been using the **context** field as a pointer to additional private data, this private data must be freed. This would normally require merely a **zone.FREE[@v.context]**; however, since the context is a **LONG UNSPECIFIED**, a **LOOPHOLE** is needed. **FreeContext** hides this **LOOPHOLE** from the implementor and does the required **zone.FREE**. It also stores the **zone** in place of **v.context**, for possible later use by **FreeStd**.

44.2.2.6 ActOn

```
ActOnProc: TYPE = PROCEDURE [data: ManagerData, action: Action]
  RETURNS [cleared: BOOLEAN ← FALSE];
```

An **ActOnProc** is provided by the manager of the selection to perform various actions on the selection. **data** is the pointer that was passed to **Set** or **SaveAndSet** and typically points to a record that describes what part of the manager's domain is currently selected. **action** indicates what action to perform (see **Action** below). An **ActOnProc** should return **cleared: TRUE** if the action resulted in the selection being cleared; that is, the manager is no longer responsible for the selection. (This should always be the case for **action: clear** and may also occur for **delete** or **clearIfHasInsert**.) **clear** should also be set for **encapsulate** to let Selection know that the manager actually supports **encapsulate**.

```
Action: TYPE = MACHINE DEPENDENT{
  clear(0), mark, unmark, delete, clearIfHasInsert, save, restore, firstFree, last(255)};
```

```
encapsulate: Action;
```

encapsulate is exported by **SelectionX**.

clear	unselects the current selection by freeing any associated private data, undoing **TIP** notification changes, etc.
mark	highlights the current selection. If the selection is already highlighted, this is a no-op.
unmark	dehighlights the current selection. If the selection is not already highlighted, this is a no-op.
delete	deletes the contents of the current selection. The selection manager may decide against actually deleting it.
clearIfHasInsert	same as **unmark** plus **clear**, but only if the insertion point (input focus) is in the selection. This action is used when a secondary selection has been completed (for copy-from); if the place to which the secondary selection is to be copied (the insertion point) is within the selection itself, the selection is cleared after obtaining its contents and before the insertion takes place.
save	unselects the current selection, but does not necessarily free any associated private data, because the selection is expected to be restored later. This action will often be a no-

	op, but the manager might need to undo a special **TIP** notifier, for example.
restore	restores a previously saved selection.
encapsulate	much like **save**, unselects the current selection but does not clear it. Always preceeded by an **unmark**. The manager is expected to do whatever is necessary to allow the selection to be operated on from the background. This may involve changing the manager data structure so that the manager procs can know that they are operating on an encapsulated selection. The manager may also lock objects down by making them "busy" or otherwise prevent them from being operated on while they are encapsulated. The manager must set clear to TRUE to indicate that the encapsulate completed successfully. See §44.2.2.7 for more discussion of encapsulated selections. **encapsulate** is exported by **SelectionX.**
firstFree	is used internally by **UniqueAction** and should not be used by clients.

Observe that, contrary to the interpretations used in the *XDE* **Selection** interface, the **clear** action does not dehighlight the selection. **Selection.Clear** (usually) does an explicit **unmark** before clearing the selection. Likewise, **save** does not imply **unmark**, nor does **restore** imply **mark**. This lets a client choose to leave a primary selection highlighted while a secondary selection is being made.

44.2.2.7 Save and Restore, Encapsulated selections

```
SaveAndSet: PROCEDURE [
  pointer: ManagerData, conversion: ConvertProc, actOn: ActOnProc,
  unmark: BOOLEAN ← TRUE]
  RETURNS [old: Saved];
```

SaveAndSet is the same as **Selection.Set** except that the existing selection, if any, is told to **ActOn[save]** rather than **ActOn[clear]**. That is, the existing selection is expected to retain any private state so that it can later be restored via **Selection.Restore**. If it subsequently turns out that the saved selection is never going to be restored, it should be given to **Selection.Discard** so that the former selection manager will have a chance to discard any associated private data. A saved selection must *always* be given eventually to either **Restore** or **Discard**; furthermore, once that has been done, the **Selection.Saved** must not be used for anything else.

It is perfectly acceptable to call **SaveAndSet** when there is no selection. If the resulting **Selection.Saved** is passed to **Selection.Restore**, it acts like **Selection.Clear**. Also, unlike for **Clear**, **ClearOnMatch**, and **Restore**, it is quite reasonable to call **SaveAndSet** with **unmark: FALSE**, thereby requesting that the saved selection remain highlighted while a secondary selection is performed. If this is done, the caller will usually wish to specify **mark: FALSE** when the saved selection is restored. Note: Calling **SaveAndSet** with **unmark: FALSE** does *not necessarily* mean that the old selection is marked. The selection manager, or some other client, might have unmarked it. The present caller is simply saying, "Do not change the highlighting on *my* account," but has no way of knowing whether the saved selection is in fact highlighted. That is why it is always up to the selection manager to decide whether **ActOn[mark]** or **ActOn[unmark]** is a no-op.

Encapsulate: PROCEDURE RETURNS [Saved];

An encapsulated selection is much like a saved selection that can be operated on from outside the notifier process. Calling this procedure tells the selection manager to encapsulate the current selection and return it so it can be operated on at a later time. Calling **Encapsulate** clears the user's selection; the implementation calls **ActOn[unmark]** followed by **Acton[encapsulate]**. If the selection manager does not support the **encapsulate** action, **Encapsulate** will raise **Error[operationFailed]**. An encapsulated selection may be passed to any of the standard requestor operations that have the eextra **manager** parameter: **ConvertX**, **EnumerateX**, and so forth. An encapsulated selection must always be cleared with **Discard**. A requestor calling **Encapsulate** should be aware that the encapsulated selection may be locked to prevent the user from operating on those objects; the requestor should not hold an encapsulated selection for any longer than necessary. **Encapsulate** is exported by **SelectionX**.

Saved: TYPE [6];

Objects of this type are created by **Selection.SaveAndSet** and **SelectionX.Encapsulate** to encapsulate a selection that is to be restored later (**SaveAndSet**) or operated on from another process (**Encapsulate**). It is opaque to prevent requestors from invoking the manager directly.

Restore: PROCEDURE [saved: Saved, mark, unmark: BOOLEAN ← TRUE];

This procedure re-institutes a previously saved selection as the current manager. The existing selection, if any, is requested to **ActOn[unmark]** (unless **unmark** is **FALSE**; see **Selection.Clear**) and then **ActOn[clear]**. The selection being restored is asked to **ActOn[restore]** and then **ActOn[mark]** (unless **mark** is **FALSE**).

Discard: PROCEDURE [saved: Saved, unmark: BOOLEAN ← TRUE];

If a client, having saved somebody else's selection (see **SaveAndSet**), determines that it should never be restored, he should call this procedure to free the associated resources. The current selection is not affected. The **ActOnProc** of the saved selection is called with **action: unmark** (unless **unmark** is **FALSE**; see **Clear**) and again with **action: clear**. Thus the **ActOnProc** must be prepared to handle these operations while the corresponding selection is saved. Callers of **Encapsulate** must call **Discard** to free the encapsulated selection. The **ActOnProc** is not called to unmark the selection in this case because **Encapsulate** always unmarks the selection first.

44.2.2.8 Miscellaneous

On all of the procedures below, use **unmark: FALSE** only if you know the area of the screen containing the selection is going to be repainted soon anyway; for example, if the window is going away.

Clear: PROCEDURE [unmark: BOOLEAN ← TRUE];

The **Clear** procedure requests that the current selection be cleared. It is equivalent to calling **ActOn[clear]**, preceded by **ActOn[unmark]** if **unmark** is TRUE. The only time **unmark** should be FALSE is if the caller knows the area of the screen containing the selection is going to be repainted soon anyway; for example, if the window containing the selection is going away.

ClearOnMatch: PROCEDURE **[pointer: ManagerData, unmark:** BOOLEAN ←TRUE**];**

It is sometimes difficult to determine if you are the manager of the current selection. The **ClearOnMatch** procedure is the same as **Clear** except that no action is taken unless **pointer** matches the **ManagerData** of the current selection. **ClearOnMatch** is equivalent to IF **Selection.Match[pointer]** THEN **Selection.Clear[unmark]**.

ActOn: PROCEDURE **[action: Action];**

The **ActOn** procedure communicates a request for an action to the manager of the current selection. (See also **UniqueAction**.) Calling **ActOn[clear]** is not recommended, since there would be a tendency to forget to **unmark** first. Use **Selection.Clear** instead.

Match: PROCEDURE **[pointer: ManagerData]** RETURNS **[match:** BOOLEAN**];**

This procedure returns TRUE if the caller is the current selection manager, which is assumed to be the case if and only if **pointer** is equal to the **ManagerData** associated with the current selection (as specified by **Set**, **SaveAndSet**, or **Restore**). **Note:** A selection manager may opt to have NIL as the **ManagerData**. In this case, the manager should not use **Match** since it would not be able to distinguish itself from other managers using NIL. However, **Match[NIL]** *always* returns FALSE if there is no selection; that is, after **Selection.Clear**.

UniqueTarget: PROCEDURE RETURNS **[Target];**

The **UniqueTarget** procedure allows a client to define its own private conversion type. It returns a new **Target** in **[firstFree..last]**. May raise **Error [tooManyTargets]**. The use of private target types severely limits the exchange of information between applications and should be avoided if possible.

UniqueAction: PROCEDURE RETURNS **[Action];**

The **UniqueAction** procedure allows an application to define its own private operations on the selection. It returns a new **Action** in **[firstFree..last]**. May raise **Error [tooManyActions]**.

44.2.3 Errors

Error: ERROR **[code: ErrorCode];**

```
ErrorCode: TYPE = {
   tooManyActions, tooManyTargets, invalidOperation,
   operationFailed, didntAbort, didntClear};
```

tooManyActions	may be raised by **UniqueAction**.

tooManyTargets	may be raised by **UniqueTarget**.
invalidOperation	raised if **Copy** or **Move** is called with a **Value** that does not implement the operation.
operationFailed	may be raised by a **ValueCopyMoveProc** if the operation is permitted but nevertheless fails, for example due to an **NSFile** error.

didntAbort and **didntClear** are never raised.

44.3 Usage/Examples

44.3.1 What Selection Is NOT

The trash bin and insertion features of the Mesa interface are not supported. If they are needed, a separate (smaller) interface should be created for them, as they do not really require the generality available for actual selections.

44.3.2 Examples of Storage Allocation for Manager's ConvertProc

Here the various storage allocation cases are discussed that arise, depending on **Target**, how the selection is maintained by the manager, etc.

- Simplest case: no storage associated with this **Target**, no copy/move.
  - Example: selection is a string in a window and **Target** = **window**.
  - Manager's **ConvertProc** should have:

    ```
    RETURN [ [value: window, ops: Selection.nopFreeValueProcs] ]
    ```

  There is nothing allocated, nothing to free, so **ops.free** is **Selection.NopFree**. It makes no sense to copy or move a window this way, so **ops.copyMove** is **NIL**.

- Slightly more complex case: no storage associated with this **Target**, allow copy/move.
  - Example: selection is a piece of a larger backing string and is maintained as an **XString.ReaderBody** and **Target** = **string**.
  - Manager's global frame:

    ```
    myValueProcs: Selection.ValueProcs ← [
        free: Selection.NopFree, copyMove: CopyMoveString ];
    SelectionData: TYPE = RECORD [ substring: XString.ReaderBody,... ];
    -- substring points at the same bytes as the backing string
    ```

  - Manager's **ConvertProc**:

    ```
    OPEN selectionData: NARROW [data, LONG POINTER TO SelectionData];
    RETURN [ [value: @selectionData.substring, ops: @myValueProcs] ];
    -- Selection will put zone into the context field.
    ```

Here the requestor points directly at the **SelectionData.substring**. The **value.value↑** cannot be changed by the requestor until after the **CopyMoveString** is called.

- Manager's **CopyMoveString**:

```
v.value ← XString.CopyReader [r: NARROW [v.value, XString.Reader],
    z: NARROW [v.context, UNCOUNTED ZONE] ];
v.ops.free ← NIL;
```

After doing the copy, **v.ops.free** is replaced with NIL, which causes **Free** to call **FreeStd**, which frees the copied **ReaderBody** and bytes. **Note: CopyReader** allocates both the **ReaderBody** and the bytes from a single allocation unit.

Note: The storage for the **SelectionData** is allocated when he **Selection.Set** is done and deallocated when **ActOn [clear]** is called.

- Typical case: some storage associated with this **Target**, allow copy/move

  - Example: selection is a piece of a larger backing string and is maintained as an **Environment.Block** and **Target = string.**

  - Manager's global frame:

```
myValueProcs: Selection.ValueProcs ← [ free: NIL, copyMove: CopyMoveString ];
SelectionData: TYPE = RECORD [ block: Environment.Block,... ];
 -- block represents the selection.
 -- block.pointer points to the backing string.
```

  - Manager's **ConvertProc**:

```
OPEN selectionData: NARROW [data, LONG POINTER TO SelectionData];
RETURN [ [
    value: zone.NEW [XString.ReaderBody ←
        XString.FromBlock [selectionData.block],
    ops: @myValueProcs] ];
    -- Selection will put zone into the context field.
    -- ops.free = NIL means that FreeStd will be called.
```

Here we allocate a **ReaderBody** that points directly into our backing string. Free will call **FreeStd**, which will free the **ReaderBody**.

  - Manager's **CopyMoveString**:

```
OPEN zone: NARROW [v.context, UNCOUNTED ZONE] ;
OPEN selectionSubstring: NARROW [v.value, XString.Reader] ;
v.value ← XString.CopyReader [ selectionSubstring, zone ];
zone.FREE [ @selectionSubstring ]; -- frees the ReaderBody
```

CopyReader copies both the **ReaderBody** and the bytes. After doing the copy, we free the **ReaderBody**. **Note:** After the copy, **Free** will still call **FreeStd**, which will free the copied **ReaderBody** and bytes.

44.3.3 Detailed Flowchart of a Selection.Convert

Following is the exact sequence of events that takes place in performing a Selection.Convert, showing what the requestor does, what the manager does, and what Selection does. Various storage allocation cases arise, depending on the Target, what the requestor wants to do, etc.Most of the cases are covered here. This will be most useful to managers, but anyone desiring an overall understanding of Selection will benefit from following these details.

- Requestor calls Selection.Convert.
- Convert calls the manager's ConvertProc. If the requestor provided a NIL zone, Convert passes Heap.systemZone.
- Manager constructs a Value, potentially allocating storage for value.value ↑ and/or for value.context ↑. value.ops may or may not be provided, depending on the selection Target and the manager. Manager returns value to Convert.
- If value.context is defaulted, Convert puts zone into value.context and returns to requestor.
- If requestor just wants to look at the converted value (not copy or move it):
  - Requestor looks at value.value ↑.
  - Requestor calls Selection.Free [@value];
  - If value.ops is NIL or value.ops.free is NIL:
    - Free calls FreeStd.
    - FreeStd recovers the zone from value.context, does a zone.FREE [@value.value], and replaces value.ops with [free: NopFree, copyMove: NIL].
  - If value.ops.free is not NIL:
    - Free calls value.ops.free [@value] (that is, the manager's ValueFreeProc).
    - The manager's ValueFreeProc recovers the zone from value.context (possibly a field in a record pointed to by value.context) and releases any resources that were allocated in the ConvertProc. This includes not only the obvious freeing of storage from the zone (zone.FREE [@value.value] and/or Selection.FreeContext [@value, zone]), but also, for example, closing or deleting any files that were created.
  - END
- If the requestor wants to move or copy the selection:
  - Requestor calls Selection.Move, Selection.Copy, or Selection.CopyMove, perhaps passing in data: LONG POINTER, which points to a destination for the move/copy.

- If value.ops is NIL or value.ops.copyMove is NIL, CopyMove raises Error [InvalidOperation]. Otherwise, CopyMove calls value.ops.copyMove [@value, {copy, move}, data] (that is, the manager's ValueCopyMoveProc).

- The manager's ValueCopyMoveProc recovers the zone from value.context, gets the destination of the move/copy from data (if appropriate), does the move or copy, calls Selection.FreeContext [@value, zone] if necessary, does a zone.FREE [@oldValue.value] if necessary. **Note:** This is freeing the original value.value, not the copied one! Now the manager can either leave value.ops.free as is, or replace value.ops.free with Selection.FreeStd (if the newly copied value was allocated from zone and zone is in value.context), or replace value.ops.free with Selection.NopFree (if there is nothing left to free).

- CopyMove replaces value.ops.copyMove with NIL to prevent another copy or move from being done.

- Requestor may retain the copied value indefinitely and/or call Selection.Free to free the copied value after using it (see above).

- END

44.3.4 Sample ConvertProc and Requestor

In this example of a simple selection manager, the selection is represented internally as a pair of indices within a single Mesa STRING. The string is inside a window. The indices designate the first character selected and the position beyond the last character selected. It isassume that there are several windows of this type, and that each contains a single string. within which selections may be made. It is also assumed that the manager's module contains a procedure TextForWindow that obtains the string associated with a window, and various other obvious utilities and signals. The procedure Select makes a new selection.

A ConvertProc is shown that implements the common targets. Observe the extremely heavy use of the defaults for the ops and context fields in the Value records. Since the Selection interface detects these defaults and applies the most common interpretations for Copy, Move, and Free, both the requestor and the manager are spared much of the coding effort.

```
-- use dynamic storage for data; global variables make save/restore awkward
myZone: UNCOUNTED ZONE = ... ;
SelectionData: TYPE = RECORD [
   w: Window.Handle, -- window containing this selection
   left, right: CARDINAL,
   marked: BOOLEAN ← FALSE];

ValueContext: TYPE = RECORD [ -- for use in Value.context fields
   zone: UNCOUNTED ZONE,
   w: Window.Handle];

Select: PROCEDURE [w: Window.Handle, left, right: CARDINAL] = {
   text: LONG STRING = TextForWindow[w];
   IF text = NIL OR left > text.length OR right NOT IN [left..text.length] THEN
```

```
      ERROR BogusSelection;
    Selection.Set[
      myZone.NEW[SelectionData ← [w, left, right]],
      ConvertSelection, ActOnSelection]};

ConvertSelection: Selection.ConvertProc = {
    <<[data: ManagerData, target: Target, zone: UNCOUNTED ZONE, info: ConversionInfo]
      RETURNS [value: Value]>>
    OPEN selectionData: NARROW [data, LONG POINTER TO SelectionData];
    WITH i:info SELECT FROM
      query => 
        FOR c: CARDINAL IN [0..LENGTH[i.query]) DO
          i.query[c].difficulty ←
            IF ~i.query[c].enumeration THEN SELECT i.query[c].target FROM
              window, string, length, position => easy,
              integer => moderate,
              ENDCASE => impossible
            ELSE --enumerated-- IF i.query[c].target = string THEN moderate
            ELSE impossible;
          ENDLOOP;
      convert =>
        SELECT target FROM
          window => RETURN[[ selectionData.w, Selection.nopFreeValueProcs]];
          length => RETURN[[zone.NEW[LONG CARDINAL ←
            selectionData.right - selectionData.left]]];
          position => RETURN[[zone.NEW[LONG CARDINAL ← selectionData.left]]];
          string, integer =>
            IF selectionData.right - selectionData.left > Selection.maxStringLength THEN
              RETURN[Selection.nullValue]
            ELSE {
              blk: Environment.Block = [LOOPHOLE[@TextForWindow[rec.w].text],
                  selectionData.left, selectionData.right];
              r: XString.ReaderBody ← XString.FromBlock[blk];
              IF target = integer THEN {
                  bad: BOOLEAN ← FALSE;
                  num: LONG INTEGER;
                  num ← XString.StringToNumber[@r
                      ! XString.InvalidNumber, XString.Overflow =>
                          {bad ← TRUE; CONTINUE}];
                  RETURN[IF bad THEN Selection.nullValue ELSE
                      [zone.NEW[LONG INTEGER ← num]]]};
              -- target = string
              RETURN[[
                  value: zone.NEW[XString.ReaderBody ← r],
                  ops: @stringOps,
                  context: zone.NEW[ValueContext ← [zone, selectionData.w]] ]]};
          ENDCASE;
      enumeration => IF target = string THEN {
        blk: Environment.Block ← [LOOPHOLE[@TextForWindow[selectionData.w].text],
          selectionData.left, TRASH];
        WHILE block.startIndex < selectionData.right DO
          block.stopIndexPlusOne ←
          MIN[block.startIndex + Selection.maxStringLength, selectionData.right];
        IF i.proc[[
          value: zone.NEW[XString.ReaderBody ← XString.FromBlock[blk]],
          ops: @stringOps,
          context: zone.NEW[ValueContext ← [zone, selectionData.w]] ]
          ].stop THEN EXIT;
        block.startIndex ← block.stopIndexPlusOne;
        ENDLOOP};
    ENDCASE;
  RETURN[Selection.nullValue]};

stringOps: Selection.ValueProcs ← [FreeString, CopyString];

FreeString: Selection.ValueFreeProc -- [v: ValueHandle] -- = {
  context: LONG POINTER TO ValueContext = v.context;
  context.zone.FREE[@v.value]; -- free the ReaderBody, but not the text bytes
  Selection.FreeContext[ v, context.zone]};

CopyString: Selection.ValueCopyMoveProc = {
  <<[v: ValueHandle, op: CopyOrMove, data: LONG POINTER]>>
  context: LONG POINTER TO ValueContext = v.context;
  old: XString.Reader = v.value;
  IF op = move THEN ERROR Selection.Error[invalidOperation];
  v.value ← XString.CopyReader[old, context.zone];
  context.zone.FREE[@old];
  Selection.FreeContext[ v, context.zone];
  v.ops.free ← NIL};

ActOnSelection: Selection.ActOnProc = {
  <<[data: ManagerData, action: Action] RETURNS [cleared: BOOLEAN ← FALSE]>>
  OPEN selectionData: NARROW [data, LONG POINTER TO SelectionData];
  SELECT action FROM
    mark, unmark => IF selectionData.marked # (action = mark) THEN
      InvertHighlighting[rec];
    save, restore => NULL; -- no special action need be taken
    delete => NULL; -- deletion is not allowed via this interface
    clearIfHasInsert => NULL; -- assume that this tool never has the insertion point
    clear => {myZone.FREE[@data]; cleared ← TRUE};
    ENDCASE};
```

Here are three sample requestors that might invoke the above manager code. The first requestor wishes to interpret the selection, if possible, as a string of digits and obtain the corresponding integer value. The second wishes to open a file whose name is the current selection. (Assume the existence of an NSFile routine that deals with XString-format file names.) The third wishes to copy the current selection to a Stream unless the selection comprises more than 10000 characters. Since copying an NSFile to an arbitrary Stream is awkward at best, it does not use Convert[file], but rather attempts to get the selection as one or more strings to send to the Stream.

```
-- Example 1: obtain selection as an integer and do something with it
num: LONG INTEGER;
ok: BOOLEAN;
[ok, num] ← Selection.ConvertNumber[integer] ;
```

```
IF ok THEN {
   << do whatever it was we wanted to do with num >>}
ELSE {
   <<report error, or ignore it>>};

-- Example 2: use current selection as name of file to open
v: Selection.Value ← Selection.Convert[string];
file: NSFile.Handle ← NSFile.nullHandle;
-- if v.value is NIL it means there's no selection, or it can't be converted to a string,
-- or the string would be so long it's not a reasonable name anyway
IF v.value # NIL THEN {
   file ← NSFile.OpenByName[v.value ! NSFile.Error => CONTINUE];
   Selection.Free[@v]};

-- Example 3: copy selection to a Stream (handle is in sH) unless length > 10000
bytes: LONG CARDINAL; ok: BOOLEAN;
v: Selection.Value;
[ok, bytes] ← Selection.ConvertNumber[length] ;
IF ok AND bytes <= 10000 THEN {
   v ← Selection.Convert[string];
   IF v.value # NIL THEN PutReader[v, sH]
   ELSE [] ← Selection.Enumerate[PutReader, string, sH]};
 ...
PutReader: Selection.EnumerationProc = {
   <<[element: Value, data: RequestorData] RETURNS [stop: BOOLEAN ← FALSE]>>
   sH: Stream.Handle = data;
   sH.PutBlock[XString.Block[element.value].block
    ! Stream.TimeOut, Volume.InsufficientSpace => {stop ← TRUE; CONTINUE}];
   Selection.Free[@element]};
```

44.3.5 Sample Use of Enumeration

In this example of the use of the enumeration facility, the user has asked to COPY or MOVE the selection to the desktop. The desktop does not particularly care what the selection is; it simply requires that it be rendered as one or more files. If the operation is a MOVE, it is better not to do it as a copy-then-delete; instead, obtain the existing files and relocate them.

```
op: Selection.CopyOrMove ← ... ; -- setting is determined by the TIP table interpretor
IF Selection.Enumerate[CopyMoveFileToDesktop, file, @op].aborted THEN { -- error -- };
.
.
.
CopyMoveFileToDesktop: Selection.EnumerationProc = {
   op: LONG POINTER TO Selection.CopyOrMove = data;
   file: LONG POINTER TO NSFile.Reference ← element.value; -- this is readonly until Copied or
   Moved
   Selection.CopyMove[@element, op ↑, handleForDesktop
         ! Selection.Error => SELECT code FROM
              -- owner will not let us have it for some reason
              invalidOperation, operationFailed => {stop ← TRUE; CONTINUE};
              ENDCASE => REJECT];
   IF stop THEN {Selection.Free[@element]; RETURN};
file ← element.value; -- the value was probably changed by Copy/Move
-- file is now a Reference to a file that is of no interest to the selection manager
<< create any associated structures necessary for keeping track of the icon >>
<< might also need to set position attributes, etc.; it would be more efficient
   to set the attributes as part of the Copy or Move, but this would probably
   require an awkward structuring of CopyMove's data parameter >>
Selection.Free[@element]; -- free the storage associated with the Reference
};
```

Here are two cases where the above code might be invoked. First, assume the selection is a set of documents in an open folder. The folder's conversion proc calls **CopyMoveFileToDesktop** once for each document, with **element** being the **NSFile.References** for the already existing files. The **ops.copyMove** provided by the folder implementation either does an **NSFile.Copy** or an **NSFile.Move** to transfer the file to the desktop directory, and updates **element.value** if necessary to refer to the new file. If the operation is a **move**, **copyMove** also reflects the deletion in the folder's window. It might also update the selection data if, for instance, the selection is represented internally as a range of positional indices within the directory.

If the selection is a set of printers in the Star directory icon, no files exist for them until they are copied to the desktop. For each printer, the conversion procedure creates a file from scratch and passes it to **CopyMoveFileToDesktop**. This time, however, **ops.copyMove** calls **NSFILE.Move** regardless of the operation requested, since it is not actually possible to remove objects from the Star directory. (Alternatively, it could call **NSFile.Move** to do a copy and raise **Error[invalidOperation]** if asked to do a move.) Meanwhile, the **ops.free** originally included with each element is **Selection.NopFree**; if the user chooses not to do anything with the printer, the Star directory enumeration code simply changes the attributes of the file to refer to the next printer in the enumeration and uses the same file again. Thus **ops.copyMove** must also set a flag indicating that a new dummy file must be created if there are any more elements in the enumeration.

The important thing to note is that, in the first example, doing a **copy** involved creating a new file, whereas in the second example it didn't. (Instead, it needed to ensure that the file not be re-used when the enumeration continued.) There was no way for the requestor to decide whether the object needed to be copied. The decision was left up to the selection manager by means of the **ops** procedure.

44.4 Index of Interface Items

Letters in parentheses indicate a description for a requestor (R) or a manager (M).

45

SimpleTextDisplay

45.1 Overview

The **SimpleTextDisplay** interface provides facilities for displaying, measuring, and resolving strings of *Xerox Character Code Standard* text. **SimpleTextDisplay** deals with text in a single font--normally the standard system font--and does not support boldface, italic, sub- and superscript, and other text properties. **SimpleTextDisplay** does not implement editable or selectable text, but it provides the building blocks that can be used to implement such things. (See **SimpleTextEdit**.)

Most clients will be interested mainly in the procedure **StringIntoWindow**, which simply displays one or more lines of text at a given location in a window.

More sophisticated clients may want to use **StringIntoBuffer**, which formats text into a special bitmap buffer rather than painting it into a window; **MeasureString**, which determines how wide a string would appear if painted into a window without actually painting it; or **FillResolveBuffer**, which computes the position of each character of an already displayed line of text.

All width values taken or returned by **SimpleTextDisplay** procedures are in terms of screen pixels (bits).

45.2 Interface Items

45.2.1 Simplest Way to Display Text

```
StringIntoWindow: PROCEDURE [
   string: XString.Reader,
   window: Window.Handle,
   place: Window.Place,
   lineWidth: CARDINAL ← CARDINAL.LAST,
   maxNumberOfLines: CARDINAL ← 1,
   lineToLineDeltaY: CARDINAL ← 0, -- default: systemFontHeight
   wordBreak: BOOLEAN ← TRUE,
   flags: BitBlt.BitBltFlags ← Display.paintFlags]
   RETURNS [lines, lastLineWidth: CARDINAL];
```

Displays **string** in **window**, starting at **place**. **place** refers to the upper-left corner of the first character. Each line is no more than **lineWidth** pixels wide, and there will be no more than **maxNumberOfLines** lines. If **wordBreak** is TRUE, **StringIntoWindow** tries to break lines between, rather than within, words. The **flags** determine what **BitBlt** function is used to place the new bits in the window; the default is to OR them into the window's existing bitmap. When a new line is started, its y-position is **lineToLineDeltaY** below the y-position of the previous line; if **lineToLineDeltaY** is defaulted to 0, each line is **systemFontHeight** pixels below the previous one. **lines** is the number of lines that were actually painted. **lastLineWidth** is the width of the last line displayed. If the string ends with a carriage return and **maxNumberOfLines** are not exceeded, then **lastLineWidth** is 0 and **lines** include an empty line following that carriage return. If the string is empty, **StringIntoWindow** returns **[lines: 0, lastLineWidth: 0]**.

StringIntoWindow always uses the standard system font, a **Flushness** of **fromFirstChar**, and a **StreakSuccession** of **fromFirstChar**. (See §45.2.4 for an explanation of **Flushness** and **StreakSuccession**.)

```
systemFontHeight: READONLY CARDINAL;
```

systemFontHeight is the height (in pixels) of the system font.

If **systemFontHeight** changes due to a runtime addition to the system font, the event **NewSystemFontHeight** will be raised with **eventData** a pointer to the **systemFontHeight**.

45.2.2 StringIntoBuffer

```
StringIntoBuffer: PROCEDURE [
   string: XString.Reader,
   bufferProc: BufferProc,
   lineWidth: CARDINAL ← CARDINAL.LAST,
   wordBreak: BOOLEAN ← TRUE,
   streakSuccession: StreakSuccession ← fromFirstChar,
   font: SimpleTextFont.MappedFontHandle ← NIL]
   RETURNS [lastLineWidth: CARDINAL, result: Result, rest: XString.ReaderBody];
```

Formats **string** into a bitmap buffer using **font** and calls **bufferProc** for each line. (See **BufferProc** below for a description of the parameters passed to **bufferProc**.) If **font** is NIL, the system font is used. **StringIntoBuffer** stops reading characters in the string and calls **bufferProc** when one of the following events occurs:

- A character whose **TextBlt** flags are **[stop: TRUE, pad: FALSE]** is encountered, such as a carriage return. **bufferProc** is called with a **result** of **stop**. The **string** passed to **bufferProc** ends with the carriage return.

- The **lineWidth** (measured in pixels) would be exceeded by formatting the next character. **bufferProc** is called with a **result** of **margin**. The **string** passed to **bufferProc** ends with the last character that did fit (if **wordBreak** is FALSE) or with the last character before the beginning of the word that did not fit (if **wordBreak** is TRUE).

- There are no more characters to be read. **bufferProc** is called with a **result** of **normal**. The **string** passed to **bufferProc** ends with the last character of the string passed to **StringIntoBuffer**.

```
Result: TYPE = {normal, margin, stop};
```

If **result** = **normal**, or **bufferProc** returns **continue** = FALSE, **StringIntoBuffer** returns the following values: **result** = the result last passed to **bufferProc**, **rest** = a substring containing characters not yet processed (**rest.offset** will be the **string.limit** last passed to **bufferProc**), **lastLineWidth** = the **dims.w** last passed to **bufferProc**.

If **result** is not **normal**, and **bufferProc** returns **continue** = TRUE, **StringIntoBuffer** continues processing the remainder of **string** and calls **bufferProc** again.

If **string** is empty, **StringIntoBuffer** returns **[width: 0, result: normal, rest: XString.nullReaderBody]** and does not call **bufferProc** at all.

```
BufferProc: TYPE = PROCEDURE [
   result: Result,
   string: XString.Reader,
   address: Environment.BitAddress,
   dims: Window.Dims,
   bitsPerLine: CARDINAL]
   RETURNS [continue: BOOLEAN];
```

A **BufferProc** is called once on each line of text processed by **StringIntoBuffer**. The procedure should return TRUE if it wants **StringIntoBuffer** to process the remaining text (and to call the **BufferProc** again). The parameters should be interpreted as follows:

result explains why **StringIntoBuffer** decided to end the current line of text:

stop	if the line ends with a carriage return character.
normal	if there are no more characters to be processed after this line. In this case, **StringIntoBuffer** ignores the **continue** boolean that the **BufferProc** returns.
margin	if the line was broken to avoid exceeding the **lineWidth** passed to **StringIntoBuffer**.

string is a substring of the string passed to **StringIntoBuffer**, which contains exactly those characters on this line. If the line ends with a carriage return, the carriage return is the last character in **string**.

address is the address of the bitmap buffer into which the current line's characters have been formatted.

dims is the dimensions of the formatted part of the bitmap buffer. **dims.h** is always equal to the height of **font** passed to **StringIntoBuffer** (or to **systemFontHeight** if **font** was NIL). **dims.w** is always < = the **lineWidth** passed to **StringIntoBuffer**.

bitsPerLine is the number of bits per bitmap line in the buffer (that is, how many bits to add to **address** to reach the beginning of the next bitmap line). It is always a multiple of 16.

Fine point: If the string passed to **StringIntoBuffer** ends in a carriage return, and the **BufferProc** returns **TRUE**, the **BufferProc** is called one last time with an empty **string** (offset and limit both equal to the passed **string.limit**), an empty bitmap (**dims.w = 0**), and **result = normal**.

45.2.3 Measure and Resolve

```
GetCharWidth: PROCEDURE [char: XChar.Character,
   font: SimpleTextFont.MappedFontHandle ← NIL]
   RETURNS [width: CARDINAL];
```

Returns the **width** of the specified character in the specified **font**. If font is **NIL**, the system font is used.

```
MeasureString: PROCEDURE [
   string: XString.Reader,
   lineWidth: CARDINAL ← CARDINAL.LAST,
   wordBreak: BOOLEAN ← TRUE,
   streakSuccession: StreakSuccession ← fromFirstChar,
   font: SimpleTextFont.MappedFontHandle ← NIL]
   RETURNS [width: CARDINAL, result: Result, rest: XString.ReaderBody];
```

MeasureString determines the number of horizontal pixels that displaying **string** in the specified **font** would take up. If **font** is **NIL**, the system font is used. If **wordBreak** is **TRUE** and the string will not fit into **lineWidth** pixels, **MeasureString** attempts to end the line between words. **result** is one of the following:

stop	If a carriage return character is encountered in the string before lineWidth pixels have been measured. In this case, **width** is the pixel width of those characters up to and including the carriage return, and **rest** begins with the first character following the carriage return.
margin	If the string will not fit within **lineWidth** horizontal pixels. In this case, **width** is the pixel width of those characters that do fit (possibly backed up to the end of the last word that entirely fits on the line, if **wordBreak** is **TRUE**), and **rest** begins with the first character that does not fit.
normal	If the string contains no carriage returns and fits entirely within lineWidth horizontal pixels. In this case, **rest** is empty.

If **string** is empty, **MeasureString** returns **[width: 0, result: normal, rest: XString.nullReaderBody]**.

```
FillResolveBuffer: PROCEDURE [
   string: XString.Reader,
   lineWidth: CARDINAL ← CARDINAL.LAST,
   wordBreak: BOOLEAN ← TRUE,
   streakSuccession: StreakSuccession ← fromFirstChar,
   resolve: ResovleBuffer,
   font: SimpleTextFont.MappedFontHandle ← NIL]
   RETURNS [width: CARDINAL, result: Result, rest: XString.ReaderBody];
```

FillResolveBuffer measures the x-offset of the left edge of each character of **string** relative to the left edge of the leftmost character and stores the measurements in the **resolve** array. The measurements are in units of pixels. The offset of the leftmost character is zero. There is one element in the resolve array for each of the bytes (not characters) of **string**. The measurement stored for each byte of **string** is the measure for the character that the byte is a part of. The measure stored for character set shift codes is that of the next actual character in the string. (For the meaning of the return values, see the description of **MeasureString**.)

The **resolve** buffer must be **string.limit-string.offset + 1** words long to avoid smashing memory.

If **string.context.suffixSize = 1** and the string contains no character set shifts (377Bs) (that is,if there is one byte per character), then:

> **resolve[0]** is assigned x-offset of the character **string.bytes[string.offset]**,
> **resolve[1]** is assigned the x-offset of the character **string.bytes[string.offset + 1]**,
> ....,
> **resolve[string.limit-string.offset-1]** is assigned the x-offset of the character **string.bytes[string.limit-1]**.

If the string does contain 377Bs, then any character set shift bytes ([377B, chset] or [377B, 377B, 0]) are assigned the same resolve value as the following character code byte.

In any part of the string that is in Stringlet16 format (2 bytes per character), both bytes of each character are assigned the same resolve value.

If a sequence of characters would be displayed as a ligature--a single graphic representing several adjacent characters--then all of those characters are assigned the same resolve value.

In all cases, **resolve[string.limit-string.offset]** is assigned the pixel width of the string-- the same value that is given to the returned value **width**.

If **string** is empty, **FillResolveBuffer** returns **[width: 0, result: normal, rest: XString.nullReaderBody]** and does not write into the **resolve** buffer at all.

ResolveBuffer: TYPE = LONG DESCRIPTOR FOR ARRAY **[0..0)** OF CARDINAL;

NewResolveBuffer: PROCEDURE **[words:** CARDINAL**]** RETURNS **[ResolveBuffer];**

Allocates a resolve buffer of the specified length for later use by **FillResolveBuffer**. Non-**SimpleTextDisplay** clients of **TextBlt** are also encouraged to obtain their resolve buffers by calling this procedure, because **SimpleTextDisplay** caches resolve buffers for efficiency.

FreeResolveBuffer: PROCEDURE **[ResolveBuffer];**

Frees a resolve buffer allocated by **NewResolveBuffer**.

45.2.4 Multinational Items

Flushness: TYPE = **{flushLeft, flushRight, fromFirstChar};**

A **Flushness** determines where to display a line of text that does not fill the entire bitmap width allotted to it. **flushLeft** places the leftmost character at the left edge of the bitmap. **flushRight** places the rightmost character at the right edge of the bitmap. **fromFirstChar** is equivalent to **flushLeft** if the first character of the text has **XChar.JoinDirection = nextCharToRight** (for example, Latin and most other alphabets); it is equivalent to

flushRight if the first character of the text has **JoinDirection = nextCharToLeft** (for example, Arabic and Hebrew letters).

```
PeekForFlushness: PROCEDURE [requestedFlushness: Flushness, string: XString.Reader]
  RETURNS [Flushness];
```

Returns a real flushness (either **flushLeft** or **flushRight**, not **fromFirstChar**) appropriate for the passed **requestedFlushness** and **string**.

```
StreakSuccession: TYPE = {leftToRight, rightToLeft, fromFirstChar};
```

```
PeekForStreakSuccession: PROCEDURE [
  requestedStreakSuccession: StreakSuccession, string: XString.Reader]
  RETURNS [StreakSuccession];
```

Returns a real streak succession (either **leftToRight** or **rightToLeft**, not **fromFirstChar**) appropriate for the passed **requestedStreakSuccession** and **string**.

45.3 Usage/Examples

The only non-Xerox Character Code that is significant to **SimpleTextDisplay** is Carriage Return. No other control characters are recognized.

All width values taken or returned by **SimpleTextDisplay** procedures are in terms of screen pixels (bits). If the client passes its own font to **SimpleTextDisplay**, its mica widths should be equal to its pixel widths. Fonts passed to **SimpleTextDisplay** should have no measurements actually in micas.

45.3.1 StringIntoWindow

```
rb: XString.ReaderBody ← XString.FromSTRING ["This is an example."L];
[] ← SimpleTextDisplay.StringIntoWindow [
  string: @rb,
  window: window,
  place: [10,10]];
```

45.3.2 StringIntoBuffer

This example shows an implementation of **StringIntoWindow** using **StringIntoBuffer**.

```
MyStringIntoWindow: PROCEDURE [
  string: XString.Reader,
  window: Window.Handle,
  place: Window.Place,
  lineWidth: CARDINAL ← CARDINAL.LAST,
  maxNumberOfLines: CARDINAL ← 1,
  lineToLineDeltaY: CARDINAL ← 0,
  wordBreak: BOOLEAN ← TRUE,
  flags: BitBlt.BitBltFlags ← Display.paintFlags]
  RETURNS [lines: CARDINAL, lastLineWidth: CARDINAL] = {
```

```
MyBufferProc: SimpleTextDisplay.BufferProc = {
   Display.Bitmap [window, [place, dims], address, bitsPerLine, flags];
   lines ← lines + 1;
   place.y ← place.y + lineToLineDeltaY;
   RETURN [continue: lines < maxNumberOfLines];
   };

IF lineToLineDeltaY = 0 THEN lineToLineDeltaY ← SimpleTextDisplay.systemFontHeight;
lines ← 0;

[lastLineWidth: lastLineWidth] ← SimpleTextDisplay.StringIntoBuffer [
   string: @rb,
   bufferProc: MyBufferProc,
   lineWidth: lineWidth,
   wordBreak: wordBreak];
};
```

45.4 Index of Interface Items

46

SimpleTextEdit

46.1 Overview

The **SimpleTextEdit** interface provides facilities for presenting short editable pieces of text, known as *fields*, to the user. The user can select, move, copy, delete, and edit the text. Such text can contain any sequence of characters supported by the *Xerox Character Code Standard.*

All the text in a **SimpleTextEdit** field is displayed in a single font. **SimpleTextEdit does not provide** multiple fonts, boldface, italics, subscript, superscript, paragraph and character properties, and other elaborate editor features.

SimpleTextEdit fields are most appropriate for short pieces of text, preferably less than 30 lines long. They are not appropriate for editing entire files, for example.

SimpleTextEdit is primarily intended to support text items in the higher-level **FormWindow** interface but is also provided as a public interface for those clients who may need it. Most clients will use **FormWindow** rather than **SimpleTextEdit**. **FormWindow** provides support for general forms, including choice, boolean, and command items. **FormWindow** also automatically adjusts the position of other fields when a text field becomes taller or shorter. The client of **SimpleTextEdit** must provide its own procedure for this.

46.1.1 Creating Fields

Fields are created by calling **CreateField**. Before creating any fields, however, a **FieldContext** must first be created by calling **CreateFieldContext**. There must be one **FieldContext** for each window that will contain **Fields**. The **FieldContext** returned by **CreateFieldContext** should be passed to **CreateField** for each field to be created. When a field is created, only the desired **Window.Dims** of the field need to be supplied.

46.1.2 Displaying a Field

A field is displayed by calling **RepaintField**. Before a field can be displayed, it must be given a **Window.Place** by calling **SetPlace**. Failure to call **SetPlace** before displaying a field results in **Error [fieldIsNoPlace]**.

46.1.3 Notifying a Field

Notifications are passed to a field by calling **TIPResults**. **SimpleTextEdit** attaches neither a window **displayProc** nor a **TIP.NotifyProc** to a window. The client provides these procedures and then calls **RepaintField** for display and **TIPResults** for notifications. If there is more than one field in a window or a single field does not occupy an entire window, the client must resolve mouse buttons to determine which field should get the notification.

46.2 Interface Items

46.2.1 FieldContext

FieldContext: TYPE = LONG POINTER TO FieldContextObject;

FieldContextObject: TYPE;

CreateFieldContext: PROCEDURE [z: UNCOUNTED ZONE, window: Window.Handle, changeSizeProc: ChangeSizeProc, font: SimpleTextFont.MappedFontHandle ← NIL] RETURNS [fc: FieldContext];

A **FieldContext** holds information that is common to all **Fields** in a given window. There must be exactly one **FieldContext** associated with any window containing **Fields**. The **FieldContext** contains such information as the fields' font, the current input focus, the field containing the current selection, and so forth.

CreateFieldContext creates a **FieldContext** for **window**, which can be later used to create individual **Fields** (see **CreateField**). Only one **FieldContext** should be created for any window. All storage associated with the **FieldContext** and its **Fields** is allocated from **z**. The **changeSizeProc** is called whenever any field's height is changed (see **ChangeSizeProc** below). All text in the **FieldContext**'s fields will be displayed with the supplied **font**. If **font** is defaulted (the usual case), the standard system font is used.

DestroyFieldContext: PROCEDURE [fc: FieldContext];

DestroyFieldContext destroys a **FieldContext**. If any of **fc**'s fields has the input focus, it clears the input focus and turns off the blinking caret. If any of **fc**'s fields contains the current selection, it clears and dehighlights the selection. **DestroyFieldContext** does *not* destroy each field. The client should either call **DestroyField** on each field *before* calling **DestroyFieldContext** or else dispose of the associated **UNCOUNTED ZONE** *after* calling **DestroyFieldContext**. The client should not call **DestroyField** after calling **DestroyFieldContext**.

SetMoveNotifyProc: PROCEDURE [fc: FieldContext, proc: MoveNotifyProc];

MoveNotifyProc: TYPE = PROCEDURE [f: Field];

A client may set a **MoveNotifyProc** and it will be called whenever part of the text of field f has been moved out of the field. Note that these two items are defined in SimpleTextEditExtra3.mesa.

46.2.2 Creating Fields

```
Field: TYPE = LONG POINTER TO FieldObject;

FieldObject: TYPE;

CreateField: PROCEDURE [
   clientData: LONG POINTER,
   context: FieldContext,
   dims: Window.Dims,
   initString: XString.Reader ← NIL,
   flushness: SimpleTextDisplay.Flushness ← fromFirstChar,
   streakSuccession: SimpleTextDisplay.StreakSuccession ← fromFirstChar,
   readOnly, password: BOOLEAN ← FALSE,
   fixedHeight: BOOLEAN ← FALSE,
   font: SimpleTextFont.MappedFontHandle ← NIL,
   backingWriter: XString.Writer ← NIL,
   SPECIALKeyboard: BlackKeys.Keyboard ← NIL]
   RETURNS [f: Field];
```

A **Field** is an area within a window that contains editable text. It is the primary object manipulated by this interface.

CreateField creates a field with appropriate attributes. The field uses the window, font, zone, and **ChangeSizeProc** of the passed **FieldContext**.

clientData is a pointer that is not interpreted but is returned by **GetClientData**. Clients may use it to associate their own data with each individual field.

dims are the initial dimensions of the field. As the field's contents change, its height may change as well (unless **fixedHeight** is TRUE). However, the height never becomes smaller than **dims.h**.

initString is the initial contents of the field, if any. **CreateField** copies the string; the caller continues to own it when **CreateField** returns.

flushness controls where to place lines of text that do not fill the entire width of the field. If **flushness** = **flushLeft**, the leftmost character is next to the field's left edge. If **flushness** = **flushRight**, the rightmost character is next to the field's right edge. If **flushness** = **fromFirstChar**, the field is **flushLeft** if its first character has **XChar.JoinDirection** = **nextCharToRight** (for example, Latin and most other alphabets), and **flushRight** if the first character has **JoinDirection** = **nextCharToLeft** (for example, Arabic and Hebrew letters).

streakSuccession indicates whether the text of the field flows **leftToRight** or **rightToLeft**. The default (**fromFirstChar**) causes the **streakSuccession** of the field to be determined from the first character in the field. Latin and most other alphabets flow **leftToRight**. Arabic and Hebrew flow **rightToLeft**.

If **readOnly** is TRUE, the user cannot change the field's contents. **SetInputFocus** is a no-op on a **readOnly** field, and any call on **TIPResults** that normally sets the input focus to this field, or change the field's contents does not do so. However, **SetValue** still works on a **readOnly** field.

If **password** is TRUE, each character of the field is displayed as a *. If a selection is made within a password field, and that selection is moved or copied, * characters are moved or copied rather than characters from the field's actual backing string. **Selection.Convert** also produces a string of " characters. The only way to access a password field's actual content is to call **GetValue**.

If **fixedHeight** is TRUE, the field's height never changes regardless of the field's content. The context's **ChangeSizeProc** is never called with this field as an argument.

font allows each field to be a different font. If **font** is NIL, then the system font is used. **Note:** This does not provide for general attributed text in **SimpleTextEdit** fields. The entire field is all the same font.

If **backingWriter** is NIL (the usual case), **SimpleTextEdit** allocates the field's backing string from the context's zone, expands it as needed, and deallocates it when the field is destroyed. If **backingWriter** is non-NIL, **SimpleTextEdit** uses it as the backing string and does not deallocate it when the field is destroyed. If **backingWriter.zone** is NIL, **TIPResults** raises **Error [noRoomInWriter]** whenever it tries to do an operation that would overflow the backing string.

SPECIALKeyboard allows a client-specified interpretation of the central keypad.

DestroyField: PROCEDURE [f: Field];

Destroys the passed field. If the field has the input focus, it clears the input focus and turns off the blinking caret. If the field contains the current selection, it clears and dehighlights the selection. **DestroyField** must not be called after the field's context has been destroyed.

GetValue: PROCEDURE [f: Field] RETURNS [XString.ReaderBody];

Returns the field's current contents. The returned string points directly into the field's backing storage; it is not copied.

SetValue: PROCEDURE [f: Field, string: XString.Reader, repaint: BOOLEAN ← TRUE];

Change the contents of the field. Copies the string, which the caller continues to own after **SetValue** returns. Repaints the field unless **repaint** is FALSE. In that case, the caller should call **RepaintField** before returning to the notifier. If the field has the input focus, it clears the input focus and turns off the blinking caret. If the field has the selection, it clears and dehighlights the selection. If **repaint** is TRUE, the field may become taller or shorter, triggering a call on the **ChangeSizeProc**.

46.2.3 Displaying a Field

RepaintField: PROCEDURE [f: Field];

Repaints the field.

SetPlace: PROCEDURE [f: Field, place: Window.Place];

Changes the window-relative location of the field. This procedure must have been called at least once before calling **GetBox**, **RepaintField**, or **TIPResults**; otherwise, calling those procedures raises **Error [fieldIsNoplace]**. Does not repaint the field. **SetPlace** is intended for two primary uses: to set the initial location of a field and to change it from within a **ChangeSizeProc** when another field gets taller or shorter.

46.2.4 Notifying a Field

TIPResults: PROCEDURE [f: Field, results: TIP.Results]
  RETURNS [tookInputFocus, changed: BOOLEAN];

Passes **results** to the specified field. The field is changed as appropriate. For example, if **results** contains a PointDown atom, the character closest to the cursor is highlighted. Details of the exact processing performed for each possible result are described below. If the field's contents are changed while processing the results, **changed** will be TRUE. If the input focus was set to this field, **tookInputFocus** will be TRUE. Both booleans start out FALSE but may become TRUE when strings or atoms are encountered in **results**. Any **TIP.Results** that change the field's contents also cause the field to be repainted; this may cause the field to become taller or shorter, triggering a call on its **ChangeSizeProc**.

If a string is encountered in **results**, the string is inserted into the field at the current insertion point. This clears the selection if the current insertion point is at either end of the selection. The passed field must be the current input focus and not **readOnly**; otherwise, the string is ignored.

The following atoms in **results** cause actions to be taken. An * indicates that the passed field must be the current input focus; if not, the atom is ignored. Unless otherwise indicated, **tookInputFocus** and **changed** remains unaffected after this atom is processed.

AdjustDown (should be preceded by a **coords** result): Extends or contracts the current selection, depending on **coords** earlier in **results**. If there is no current selection, creates one extending from the current insertion point to a place determined by **coords**. This is a no-op if the passed field is not the current input focus or selection.

AdjustMotion (should be preceded by a **coords** result): Same effect as **AdjustDown**, although a different algorithm is used to determine which endpoint of the selection is being moved.

BackSpace*: If the field is not **readOnly**, deletes the character before the insertion point and sets **changed** to TRUE. This clears the selection if the current insertion point is at either end of the selection.

BackWord*: If the field is not **readOnly**, deletes the word before the insertion point and sets **changed** to TRUE. This clears the selection if the current insertion point is at either end of the selection. If the field is a password field, acts like a **BackSpace**.

CopyDown: Calls **TIPStar.SetMode [copy]**.

CopyModeDown (should be preceded by a **coords** result): If the field is not **readOnly**, places the caret at an appropriate place in the field, depending upon **coords** earlier in results, but leaves the selection alone. **tookInputFocus** will be **TRUE**. If the field is **readOnly**, this is a no-op and **tookInputFocus** is unchanged.

CopyModeMotion (should be preceded by a **coords** result): Same effect as **CopyModeDown**.

CopyModeUp*: If the field is not **readOnly**, inserts the current selection at the current insertion point, sets the selection to be the newly inserted text, and calls **TIPStar.SetMode [normal]**. If the selection is not empty, repaints the field and sets **changed** to **TRUE**. If the field is **readOnly**, this is a no-op and **changed** remains unaffected.

DeleteDown: Calls **Selection.ActOn [delete]**. **changed** becomes **TRUE**.

MoveDown: Calls **TIPStar.SetMode [move]**.

MoveModeDown (should be preceded by a **coords** result): Same effect as **CopyModeDown**.

MoveModeMotion (should be preceded by a **coords** result): Same effect as **CopyModeDown**.

MoveModeUp*: Same effect as **CopyModeUp**, except that it does a **Selection.ActOn [delete]** on the current selection before setting the selection to be the newly inserted text. Note that if the current selection is in a **readOnly** field, no deletion occurs, and it acts exactly like a **CopyModeUp**.

NewLine*: If the field is not **readOnly**, inserts an **Ascii.CR** at the current caret position. This clears the selection if the current insertion point is at either end of the selection. **changed** will be **TRUE**. If the field is **readOnly**, is a no-op and **changed** is unaffected.

NewParagraph*: Same effect as **NewLine**.

PointDown (should be preceded by a **coords** result and a **time** result): Sets the current selection to be in the passed field. The location of the selection depends upon **coords** earlier in **results**; the extent (character, word, paragraph) depends on its current extent and **time** earlier in **results**. **tookInputFocus** is **TRUE** unless the field is **readOnly**.

PointMotion (should be preceded by a **coords** result): Moves the current selection within the field. If the current selection is not in the field, it sets it there. The location of the selection depends upon **coords** earlier in **results**. The extent of the selection (character, word, paragraph) remains unchanged. **tookInputFocus** is **TRUE** unless the field is **readOnly**.

PointUp (should be preceded by a **time** result): Sets the last-click time,which determines whether a subsequent **PointDown** represents a multiple click.

Stop: Calls **TIPStar.SetMode [normal]**.

46.2.5 Miscellaneous Get and Set Procedures

GetBox: PROCEDURE [f: Field] RETURNS [box: Window.Box];

Returns the box (dimensions and place) currently occupied by f. **box.place** is relative to the field's window and is always the last value passed to **SetPlace**. Raises **Error [fieldIsNoplace]** if **SetPlace** has never been called on this field.

GetClientData: PROCEDURE [f: Field] RETURNS [clientData: LONG POINTER];

Returns the **clientData** that was passed to **CreateField**.

GetFieldContext: PROCEDURE [f: Field] RETURNS [FieldContext];

Returns the field context that was passed to **CreateField**.

GetFlushness: PROCEDURE [f: Field] RETURNS [SimpleTextDisplay.Flushness];

Returns the current **Flushness** of f.

GetFont: PROCEDURE [f: Field]
RETURNS [SimpleTextFont.MappedFontHandle];

GetInputFocus: PROCEDURE [fc: FieldContext] RETURNS [Field];

If some field associated with **fc** has the input focus, it returns that field; otherwise, it returns NIL.

GetCaretPlace: PROCEDURE [context: FieldContext]
RETURNS [place: Window.Place];

If any field in the **FieldContext** contains the current type-in point, this procedure returns the location of that point. If not, place=[-1,-1]. This is useful for determining that the window must be scrolled to make the caret visible to the user.

GetReadOnly: PROCEDURE [f: Field] RETURNS [BOOLEAN];

Returns the current value of **readOnly** for f.

GetStreakSuccession: PROCEDURE [f: Field] RETURNS [SimpleTextDisplay.StreakSuccession];

Returns the current **StreakSuccession** of f.

GetWindow: PROCEDURE [fc: FieldContext] RETURNS [window: Window.Handle];

Returns the window that was passed to **CreateFieldContext**.

GetZone: PROCEDURE [fc: FieldContext] RETURNS [UNCOUNTED ZONE];

Returns the UNCOUNTED ZONE that was passed to **CreateFieldContext**.

SetDims: PROCEDURE [f: Field, dims: Window.Dims];

SetDims sets the dimensions for f.

SetFixedHeight: PROCEDURE [f: SimpleTextEdit.Field, fixedHeight: BOOLEAN];

Allows setting the fixed-height attribute for a field.

SetFlushness: PROCEDURE [f: Field, new: SimpleTextDisplay.Flushness] RETURNS [old: SimpleTextDisplay.Flushness];

Changes the field's flushness and returns the old flushness. Does not repaint the field.

SetFont: PROCEDURE [f: Field, font: SimpleTextFont.MappedFontHandle ← NIL];

If **font** = NIL, the system font is used.

SetInputFocus: PROCEDURE [f: Field, beforeChar: CARDINAL ← CARDINAL.LAST];

Sets the current input focus to be in this field and places the blinking caret before the specified character. If **beforeChar** is 0, puts the caret before the first character; if it is CARDINAL.LAST or otherwise larger than the length of the backing string, the caret is placed after the last character in the field. Does not affect the current selection.

SetReadOnly: PROCEDURE [f: Field, readOnly: BOOLEAN] RETURNS [old: BOOLEAN];

Changes the field's **readOnly** attribute and returns its old value. If this field has the input focus and **readOnly** is TRUE, it clears the input focus and turns off the blinking caret. If this field has the selection and **readOnly** is FALSE and **old** is TRUE, it sets the input focus to this field and places the caret after the last character in the selection.

SetSelection: PROCEDURE [f: Field, firstChar: CARDINAL ← 0, lastChar: CARDINAL ← CARDINAL.LAST];

Sets the current selection to be in this field, covering the specified range of characters. If **firstChar** is 0, the selection begins with the first character of the field. If **lastChar** is CARDINAL.LAST or otherwise larger than the length of the backing string, the selection extends to the end of the string. Highlights the selection if it is not empty. Does not affect the input focus or caret.

SetStreakSuccession: PROCEDURE [f: Field, new: SimpleTextDisplay.StreakSuccession] RETURNS [old: SimpleTextDisplay.StreakSuccession];

Changes the field's **StreakSuccession** and returns the old **StreakSuccession**. Does not repaint the field.

SetLosingFocusProc: PROCEDURE [fc: FieldContext, proc: LosingFocusProc];

LosingFocusProc: TYPE = PROCEDURE [f: Field];

SetLosingFocusProc sets the **LosingFocusProc** for **fc**. **proc** is called whenever a field in **fc** loses the input focus. **f** is the field that is losing the input focus. This allows the client to undo things that were done when the input focus was set, such as clear softkeys. These items are defined in SimpleTextEditExtra.mesa.

SetUseFinalRenderingForms: PROCEDURE [fc: FieldContext, useFinalRenderingForms: BOOLEAN];

GetUseFinalRenderingForms: PROCEDURE [fc: FieldContext] RETURNS [useFinalRenderingForms: BOOLEAN];

useFinalRenderingForms has meaning for languages in which characters are rendered differently at the end of a word than in the middle of a word, such as in Arabic. These items are defined in SimpleTextEditExtra2.mesa.

SetCursorKeys: PROCEDURE [field: Field, enable: BOOLEAN] RETURNS [oldEnable: BOOLEAN];

SetCursorKeys is for clients that want to enable or disable cursor movement keys in a field. Note this is defined in SimpleTextEdit4.mesa.

LineFromY: PROCEDURE [f: Field, y: INTEGER] RETURNS [line: CARDINAL, lineReaderBody: XString.ReaderBody];

LineFromY will find the line of text which is **y** pixels down from the top of the field f, and will return the **line** number and the ReaderBody describing the line. **lineReaderBody** points directly into the SimpleTextEdit field's backing string and should therefore NOT be freed by the client. Note this is defined in SimpleTextEdit5.mesa.

AppendReader: PROCEDURE [f: Field, string: XString.Reader, repaint: BOOLEAN ← TRUE];

AppendReader is used by the client to a append **string** to the end of the backing store for field f. The field does not need to have the input focus, and the selection and the input focus remain unchanged. If **repaint** is FALSE, the affected area will be invalidated but not validated. If **repaint** is TRUE, the display will reflect all the changes after returning from **AppendReader**. Note this is defined in SimpleTextEdit5.mesa.

46.2.6 ChangeSizeProc

ChangeSizeProc: TYPE = PROCEDURE [f: Field, oldHeight, newHeight: INTEGER, repaint: BOOLEAN];

Each **FieldContext** has a **ChangeSizeProc** associated with it. This procedure is called whenever any of its fields is redisplayed and the number of lines of text being displayed has changed. It may be called as a result of calling either **RepaintField**. **TIPResults**, or **SetValue**. The client is expected to update any affected data structures (such as the **Window.Place** of other fields) and then optionally repaint any part of the window that is invalid. (There are two exceptions: the **ChangeSizeProc** is never called on a field for which **CreateField** was called with **fixedHeight** = TRUE, and it is not called if both the old and new number of text lines require fewer vertical pixels than the height **dims.h** that was specified to **CreateField**.)

The **oldHeight** and **newHeight** parameters are in vertical pixels. **inDisplayProc** is TRUE if the **ChangeSizeProc** is being called as a result of calling **RepaintField** with **repaint** = TRUE (that is, is being called indirectly by **Window.Validate**).

If **repaint** is TRUE, the **ChangeSizeProc** should not do a **Window.Validate**, because this would cause undesirable recursion.

46.2.7 Errors

Error: ERROR [type: ErrorType];

ErrorType: TYPE = {fieldIsNoplace, noRoomInWriter, lastCharGTfirstChar};

Error [fieldIsNoplace] is raised by **GetBox**, **RepaintField**, and **TIPResults** if **SetPlace** has never been called on the passed field. **Error [noRoomInWriter]** is raised by **CreateField**, **SetValue**, and **TIPResults** if a non-NIL **backingWriter** was passed to **CreateField**, the **backingWriter** has a NIL zone, and the desired operation would overflow the string.

46.3 Usage/Examples

46.3.1 Selection Management

If certain atoms (**PointDown**, **PointMotion**, **AdjustDown**, **AdjustMotion**, **CopyModeUp**, **MoveModeUp**) are in the **TIP.Results** passed to **TIPResults**, **SimpleTextEdit** may become the manager of the current selection. The procedure **SetSelection** also causes **SimpleTextEdit** to manage the curent selection.

While **SimpleTextEdit** is managing the current selection, it supports conversions to the following **Selection.Targets**: **shell**, **subwindow**, **length**, and **string**. It also supports **Selection.Enumerate** with a target of **string**.

SimpleTextEdit implements the following **Selection.Actions**:**mark**, **unmark**, **clear**, **delete**, **clearIfHasInsert**, **restore**, and **save**. All other **Actions** are ignored.

Selection.ActOn [delete] automatically repaints the field that contained the current selection; the field may become taller or shorter, triggering a call on its **ChangeSizeProc**. **Selection.ActOn [delete]** is a no-op if the current selection is in a **readOnly** field.

46.4 Index of Interface Items

47

SimpleTextFont

47.1 Overview

The **SimpleTextFont** interface provides access to the default system font that is used to displayViewPoint's text, such as the text in menus, the attention window, window name stripes, containers, property sheet text items, and so forth. This interface is a specialization of the regular font management subsystem.

47.2 Interface Items

47.2.1 System Font

MappedFontDescriptor: TYPE;

MappedFontHandle: TYPE = LONG POINTER TO MappedFontDescriptor;

MappedFontDescriptor is an opaque type that contains all of the information about a font. (All metrics, including the width of each character, are in screen dots, not micas.)

MappedDefaultFont: PROCEDURE RETURNS [MappedFontHandle];

MappedDefaultFont returns the client a handle onto the system default font. May raise **FontNotFound** or **Problem[badFont]**. The implementation of **SimpleTextFont** expects that the default font is available in the system file catalog, with the name System.NovaFont.

```
MappedFont: PROCEDURE [name: XString.Reader← NIL]
  RETURNS [ MappedFontHandle];
```

MappedFont returns a handle onto the named system font. The file must be a child of the system file catalog. Supplying NIL is the equivalent of calling **MappedDefaultFont**. May raise **FontNotFound** or **Problem[badFont]**.

```
MappedFontFromReference: PROCEDURE [NSFile.Reference]
  RETURNS [ MappedFontHandle];
```

MappedFontFromReference returns a handle onto the specified font file. May raise **Problem[badFont]**. This is defined in SimpleTextFontExtra2.mesa.

```
UnmapFont: PROCEDURE [MappedFontHandle];
```

Unmaps a font that was mapped with **MappedFont** or **MappedFontFromReference**. **UnmapFont** is defined in SimpleTextFontExtra.mesa.

47.2.2 Client-Defined Characters

```
AddClientDefinedCharacter: PROCEDURE [
   width, height: CARDINAL,
   bitsPerLine: CARDINAL,
   bits: LONG POINTER,
   offsetIntoBits: CARDINAL ← 0 ]
   RETURNS [XString.Character];
```

AddClientDefinedCharacter adds the client's bitmap to the system font as a new character and returns the 16-bit value of the character position it is assigned. **offsetIntoBits** is a byte offset. The new character's **TextBlt** flags indicate that it is neither a stop nor a pad character. At start-up time, at least 100 slots are available for these new characters. [0,26] normally displays as the blob character. May raise **Problem[clientCharacterBitsExhausted]** or **Problem[clientCharacterCodesExhausted]**. If RESUMEd, the character [0,26] is returned.

The *Xerox Character Code Standard* sets aside a block of character codes for user definition. (See the *Xerox Rendering Code Standard*, XSIS 068208, page 6.) In Star, it is often useful to include a small picture, for example, a 13x13 icon drawing, within a message or other text.

The **AddClientDefinedCharacter** procedure provides a convenient way of presenting such small pictures within formatted system text. You create a character for the picture, say in initialization code, and then simply use that (16-bit) character within ordinary text sequences, such as window titles.

47.2.3 Signals and Errors

```
FontNotFound: SIGNAL [name: XString.Reader];
```

If **FontNotFound** is resumed, the system font is used.

```
Problem: SIGNAL [code: ProblemCode];

ProblemCode: TYPE =
 {badFont, clientCharacterCodesExhausted, clientCharacterBitsExhausted};
```

47.3 Usage/Examples

SimpleTextFont is a specialization of the regular font management subsystem.

The font file format is easily parsed, it can be mappable into read-only virtual memory for use, and it can be extended. A single file defines the bitmaps for the Xerox characters in one font face and one font size, such as Bodoni Italic 10. Fine point: In the case in which several different font face/sizes have the same pictures, as can occur with some printwheel fonts, use the same file for more than one font/face. This subject is outside the scope of this specialized interface, because we are only dealing with one font.

The font file begins with a header that identifies the font and describes the subsequent sections. Each subsequent section then contains **TextBlt**-style information about one character set's characters. Fine point: Descriptions of the font management subsystem and the ViewPoint font format are to be found elsewhere.

47.3.1 Adding a Client-Defined Character

The following example creates a small (13x13) icon and displays it as part of a string:

```
myBits: ARRAY [0..13) OF WORD ← [--some bits--];
wb: XString.WriterBody ← XString.WriterBodyFromSTRING[" is an icon."];
smallPicture:XString.Character ← SimpleTextFont.AddClientDefinedCharacter [
    width: 13,
    height: 13,
    bitsPerLine: 16,
    bits: @myBits,
    offsetIntoBits: 0];

XString.AppendChar[to: @wb, c: smallPicture];

[] ← SimpleTextDisplay.StringIntoWindow [
 string: XString.ReaderFromWriter[@wb],
 window: window,
 place: place];
```

47.3.2 Acquiring the System Font

The following example acquires a handle to the system font:

```
systemFont: SimpleTextFont.MappedFontHandle = SimpleTextFont.MappedFont[];
```

47.3.3 New System Font

If the system font changes during runtime the event **NewSystemFont** will be raised with the **eventData SimpleTextFont.MappedFontHandle**.

47.4 Index of Interface Items

48

SoftKeys

48.1 Overview

The **SoftKeys** interface provides for client-defined function keys designated to be the isolated row of function keys at the top of the physical keyboard. It also provides a **SoftKeys** window whose "keytops" may be selected with the mouse to simulate pressing the physical key on the keyboard. Such a window will be displayed on the user's desktop whenever an interpretation other than the default **SoftKeys** interpretation is in effect. (The default is assumed to be the functions inscribed on the physical keys.)

48.2 Interface Items

48.2.1 Data Structures for SoftKey Labels

```
numberOfKeys: CARDINAL = . . . ; -- This number is dependent on the physical keyboard.
```

Represents the number of keys in the soft key row. **Important:** in SoftKeys.mesa, **numberOfKeys** is defined as a constant 8. This constant should not be used. Use the SoftKeysExtra.mesa **numberOfKeys** public variable.

```
LabelRecord: TYPE = RECORD [
   unshifted: XString.ReaderBody ← XString.nullReaderBody,
   shifted: XString.ReaderBody ← XString.nullReaderBody];
```

LabelRecord provides a record of two **XString.ReaderBody** arrays so that both the shifted and unshifted key meanings may be labeled. It is expected that any individual key will have either a single **unshifted** label centered on the picture of the appropriate keytop, or both **shifted** and **unshifted** labels painted in two lines on the keytop, or no label at all (**XString.nullReaderBody** for both **shifted** and **unshifted**).

```
Labels: TYPE = LONG DESCRIPTOR FOR ARRAY OF LabelRecord;
```

Client-owned array of strings to be used as labels on the **SoftKeys** virtual keytops. The **SoftKeys** procedures expect an array of up to **numberOfKeys** **LabelRecord**'s at a time.

Clients should see to it that string deallocation does not occur between calls to create and delete a **SoftKeys** instance.

Bitmaps may be specified for individual labels by using **SimpleTextFont.AddClientDefinedCharacter**. The current **SimpleTextFont** implementation has a somewhat limited number of available slots for client-defined keys. (See the **SimpleTextFont** interface for more information.)

48.2.2 Creating and Deleting SoftKeys

```
Push: PROCEDURE [
   table: TIP.Table ← NIL,
   notifyProc: TIP.NotifyProc ← NIL,
   labels: Labels← NIL,
   highlightedKey: CARDINAL ← nullKey,
   outlinedKey: CARDINAL ← nullKey]
   RETURNS[window: Window.Handle];
```

Push installs the **SoftKeys** interpretation in the following way: (1) If there is a non-NIL **table**, it is installed in the TIP watershed (see **TIPStar**); (2) if there is a non-NIL **notifyProc**, it is attached to **NormalKeyboard.TIP**. The latter has the effect of passing all productions matched in **NormalKeyboard.TIP** to your **notifyProc**. (See Appendix A for a complete listing of **NormalKeyboard.TIP**.)

A **SoftKeys** window is displayed by using **labels** to "inscribe" the keytop pictures with the new names of the keys. Both the shifted and unshifted state of a key may be labeled. If only the unshifted state is relevant, the shifted state may be defaulted to **XString.nullReaderBody**. If there are fewer strings than keytops needing them, the remaining keys are left blank. Extra strings are ignored. Fine point: Bitmaps may be placed on the keytops by using **SimpleTextFont.AddClientDefinedCharacter**. Storage for the **label** strings is the responsibility of the client. Care should be taken to ensure that this storage is kept intact between a **Push** and a **Remove** of any given **SoftKeys** interpretation.

outlinedKey and **highlightedKey** appear highlighted and/or outlined when the window is initialized. The default is no outlining or highlighting. Key values assume zero indexing [0..**SoftKeysExtra.numberOfKeys**). (That is, the key marked Center is key 0, Bold is key 1,and so forth.)

Push returns a handle to the client's **SoftKeys window**. **Note:** There may be more than one **SoftKeys** window, with each client holding the handle to his own. The last **Pushed** interpretation is the one in effect until it is **Removed** or superseded by another **Push**.

```
Remove: PROCEDURE [window: Window.Handle];
```

The **Remove** procedure removes the **SoftKeys** interpretation and associated **SoftKeys window**. The client is responsible for removing its **SoftKeys** interpretation when it relinquishes control of the selection/input focus [see **Selection** interface descriptions of **ActOn** and **Clear**] or the user's attention (as in the case of the keyboard and font keys). A **SoftKeys window** and its associated **SoftKeys** interpretation constitute a unique **SoftKeys** instance. Any **SoftKeys** instance may be removed from the stack of **SoftKeys** instances in an order other than the order pushed.

Attempts to **Remove** without the corresponding valid **window** handle from a **Push** result in the error **InvalidHandle**.

Fine point: **Remove**, rather than **Pop**, was chosen to describe the function opposite **Push** to clarify that this is not a true stack. While **Push**, as the name implies, acts on the top of the stack, **Remove** does not. It is possible to **Remove** a **SoftKeys** window from other than the top of the stack.

```
Swap: PROCEDURE [
   window: Window.Handle,
   table: TIP.Table ← NIL,
   notifyProc: TIP.NotifyProc ← NIL,
   labels: Labels← NIL,
   highlightedKey: CARDINAL ← nullKey,
   outlinedKey: CARDINAL ← nullKey];
```

The **Swap** procedure is a way to exchange **SoftKeys** interpretations without changing the **SoftKeys** instance. Current examples of use include the keyboard key implementation where pressing the More key brings up another group of **SoftKeys** choices. It is strongly suggested that a client utilizing a More key place it on the first soft key (the key marked CENTER on the physical keyboard) for a consistent user interface.

At the time when *no* **SoftKeys** interpretation is desired, a single **Remove** corresponding to the original **Push** is expected. Any number of **Swaps** may occur in between. Attempts to **Swap** without the corresponding valid **window** handle from a **Push** result in the error **InvalidHandle**.

48.2.3 Highlighting and Outlining a SoftKeys Keytop Picture

```
HighlightThisKey: PROCEDURE [
   window: Window.Handle
   key:CARDINAL ← nullKey];

OutlineThisKey: PROCEDURE [
   window: Window.Handle,.
   key: CARDINAL ← nullKey];
```

These procedures are provided for those clients where permanent highlighting and/or outlining of certain soft keys is desired. (Do not confuse these procedures with the highlighting done when a key is selected with the mouse. That highlighting is done without client participation.) The first parameter, **window**, refers to the client's **SoftKeys** window returned from a **Push**. The **CARDINAL** corresponds to the key (zero indexing) to be outlined or highlighted whenever the chosen key changes. A **key** value of **nullKey** undoes a key that is currently highlighted (or outlined). A number other than **nullKey** or [0..**SoftKeysExtra.numberOfKeys**) results in NoOp.

Attempts to call **HighlightThisKey** or **OutlineThisKey** without a valid **handle** from a **Push** result in the error **InvalidHandle**.

```
nullKey: CARDINAL = LAST[CARDINAL];
```

A default value meaning no key, to be used for **outlinedKey** and **highlightedKey**.

48.2.4 Retrieving Information About a SoftKeys Window Instance

```
Info: PROCEDURE [
   window: Window.Handle]
   RETURNS [
   table: TIP.Table,
   notifyProc: TIP.NotifyProc,
   labels: Labels,
   highlightedKey:CARDINAL,
   outlinedKey:CARDINAL];
```

The **Info** procedure returns information relevant to the **SoftKeys** instance related to **window**. If the **window** handle is not valid, the error **InvalidHandle** is returned.

48.2.5 Errors

```
InvalidHandle: ERROR;
```

This error is raised if the **SoftKeys window** handle passed to **Remove**, **Swap**, **Info**, **HighlightThisKey**, or **OutlineThisKey** is invalid.

48.3 Usage/Examples

48.3.1 Graphics Example

```
--When the selection is such that the graphics code takes control,
-- the initial graphics code should put up the graphics soft keys:
   graphicsSoftKeysWindow ← Push[
     table: graphicsSoftKeysTIPTable,
    labels: graphicsSoftKeyLabels];

--where the core of the graphics TIP.Table looks something like:
   --left-side values are defined in the LevelIVKeys interface
   SELECT TRIGGER FROM
      CenterDown => Stretch;
      BoldDown => Magnify;
      ItalicsDown => Grid;
      CaseDown, UnderlineDown => Line;
      DbkUnderlineDown, SuperscriptDown => Curve;
      StrikeoutDown, SubscriptDown => Join;
      SuperSubDown, SmallerDown => Top;
      ENDCASE;

--and part of the graphics TIP.NotifyProc resembles the following:
   --left-side values are the atom results from the TIP.Table
   atom => SELECT result FROM
      Stretch => DoMyStretchRoutine[];
      Magnify => DoMyMagnifyRoutine[];
      Grid => DoMyGridRoutine[];
      Line => DoMyLineRoutine[];
      Curve => DoMyCurveRoutine[];
    Join => DoMyJoinRoutine[];
    Top => DoMyTopRoutine[];
    ENDCASE;
```

--When graphics loses the selection, it must clear away its SoftKeys interpretation.
```
  Remove[graphicsSoftKeysWindow];
```

A client using More as one of its soft keys handles it in hits s TIP.Tables and TIP.NotifyProc:

```
  TIP.Table entry:
    Center Down => More;

  NotifyProc entry:
    More => Swap[
    window: mySoftKeysWindow,
    table: myNextSoftKeysTIPTable,
    labels: myNextSoftKeyLabels,
    highlightedkey: 1];
```

This entry results in an exchange of the client's last SoftKeys interpretation for the next one specified, (namely, the installation of the new TIP.Table and new labels on the keytops.) The second key (bold on the physical keyboard) is highlighted in the SoftKeys window. The outlinedKey parameter has been left blank. This defaults to nullKey, in which case no key will be outlined.

48.3.2 Keyboard Manager Example

This client's (Keyboard Manager) SELECT arm does the right thing for both the 8010 and 6085 workstation keyboards.

```
  atom => SELECT z.a FROM
    CenterDown => IF more THEN SoftKeys.Swap [] ELSE InstallKeyboard [label1];
    BoldDown => InstallKeyboard [label2];
    ItalicsDown => InstallKeyboard [label3];
    CaseDown, UnderlineDown => InstallKeyboard [label4];
    DbkUnderlineDown, SuperscriptDown => InstallKeyboard [label5];
    StrikeoutDown, SubscriptDown => InstallKeyboard [label6];
    SuperSubDown => InstallKeyboard [label7]; -- No label7 if the machine is an
                                                                   8010.
    DbkSmallerDown => InstallKeyboard [label8]; -- No label8 for 8010 either.
    MarginsDown, SmallerDown => ShowKeyboard [];
    FontDown, DefaultsDown => SetKeyboard [];
```

If the user presses the MarginsDown on a 6085 or the SmallerDown on an 8010, he has actually invoked the soft key that is labeled SHOW in the soft keys window visible on the screen.

48.4 Index of Interface Items

49

StarDesktop

49.1 Overview

The **StarDesktop** interface provides access to assorted facilities related to the ViewPoint desktop.

49.2 Interface Items

49.2.1 General

```
AddReferenceToDesktop: PROCEDURE [
   reference:NSFile.Reference,
   place:Window.Place ← nextPlace];

nextPlace: Window.Place = [-1, -1];
```

Adds an icon to the desktop. The file (**reference**) must be a child of the desktop file (see **GetCurrentDesktopFile** below.) If there is already an icon at **place**, the next available place is used.

```
GetPlaceFromReference: PROCEDURE [ref: NSFile.Reference]
   RETURNS [Window.Place];
```

This returns the location of an icon on the desktop. It may be used with **AddReferenceToDesktop** to place an icon near another icon by passing the return value from **GetPlaceFromReference** to **AddReferenceToDesktop**. **AddReferenceToDesktop** places the new icon at the next available spot after the place passed in.

```
SelectReference: PROCEDURE [reference: NSFile.Reference]
   RETURNS [ok: BOOLEAN];
```

Selects the icon associated with the specified reference. **SelectReference** returns FALSE if selection fails (for example, if the reference is not found on the desktop). Each call to **SelectReference** will add that reference to the selection (like doing an extended selection with adjust). To select a single icon, call **Selection.Clear** followed by a **SelectReference**.

GetWindow: PROCEDURE RETURNS [window: Window.Handle];

Returns the desktop window (that is, the root window for ViewPoint).

GetShellFromReference: PROCEDURE [ref: NSFile.Reference]
RETURNS [sws: StarWindowShell.Handle];

If an icon has a shell currently opened, **GetShellFromReference** returns this shell.

CreateDesktop: PROCEDURE [name: XString.Reader]
RETURNS [fh: NSFile.Handle];

Creates a new desktop directory and returns a handle to it. **name** is typically a fully qualified three-part user name. It is used by logon plug-in clients, a friends-level facility (as opposed to a public facility).

GetCurrentDesktopFile: PROCEDURE RETURNS [NSFile.Reference];

Every available desktop is an **NSFile** with **attribute.isDirectory = TRUE**. Desktops have children that are also **NSFiles** and show up as icons on the desktop (see the *ViewPoint Programmer's Guide*, chapter 3, for more information). **GetCurrentDesktopFile** returns the **NSFile.Reference** for the desktop **NSFile** that is currently installed and displayed to the user.

GetNextUnobscuredBox: PROCEDURE [height: INTEGER] RETURNS [Window.Box];

GetNextUnobscuredBox returns the next available vertical segment of the desktop window of height, **height**, and **width** the width of the desktop. This is intended for such things as the Attention window and the typing feedback window for JStar. There is no guarantee that the box returned will be visible, i.e., the client must ensure that the returned box is within the desktop window.

MakeBusy: PROCEDURE [ref: NSFile.Reference]
RETURNS [BusyIcon.BusyStatus];

MakeUnbusy: PROCEDURE [ref: NSFile.Reference]
RETURNS [BusyIcon.BusyStatus];

IsBusy: PROCEDURE [ref: NSFile.Reference]
RETURNS [yes: BOOLEAN];

These procedures support the BusyIcon interface by allowing the client to make icons busy and unbusy and determine if an icon on the desktop is busy. See the **BusyIcon** chapter for details on busy icons. These interfaces are defined in **StarDesktopExtra**.

SetDisplayBackgroundProc: PROCEDURE [PROCEDURE [Window.Handle]];

SetDisplayBackgroundProc allows a client to change the procedure that displays the background for the desktop.

49.2.2 Atoms

Several **ATOMs** are exported by the **StarDesktop**:

attemptingLogoff	"AttemptingLogoff": Event just before logoff. Can be vetoed. Gives clients a chance to veto logoff.
desktopWindowAvailable	"DesktopWindowAvailable": Event notified when the desktop window has been initialized and inserted into the window tree. Cannot be vetoed.
fullUserName	"FullUserName": This atom is to be used with **AtomicProfile**.
newIcon	"NewIcon": Event notified when an icon has been added to the desktop, either by user copy/move, or by a client call to **AddReferenceToDesktop**.
logoff	"Logoff": Event occurs after logoff. Cannot be vetoed.
logon	"Logon": Event notified after successful ViewPoint logon. Cannot be vetoed. EventData is **NSFile.Handle** for the desktop file.
userPassword	"UserPassword": Also to be used with **AtomicProfile**.

49.3 Usage/Examples

49.3.1 Adding a Reference to the Desktop

```
BuildFile: PROCEDURE [--parms--] = {
   reference: NSFile.Reference ←
      InitializeFile [parent: StarDesktop.GetCurrentDesktopFile[] ]; -- local proc
   place:Window.Place ← [...];
   .
   .
   .
   StarDesktop.AddReferenceToDesktop [reference, place];
   };
```

49.4 Index of Interface Items

50

StarWindowShell

50.1 Overview

StarWindowShell allows a client to create a Star-like window. A **StarWindowShell** window has a header that contains a title, commands, and pop-up menus. It may have both horizontal and vertical scrollbars. It has interior window space that may contain anything the client desires (see Figure 50.1.) **StarWindowShell** also supports the notion of "opening within." The client is insulated from the implementation-specific details of exactly how these features are represented on the display as well as how windows are arranged on the screen (for example, whether they overlap).

50.1.1 Client overview

A **StarWindowShell** is a window (see **Window** interface) that is a child of the desktop window. A **StarWindowShell** has an interior window that is a child of the **StarWindowShell** and is exactly the size of the available window space in the shell (that is, the **StarWindowShell** minus its borders and header and scrollbars). The interior window may have child windows created by the client. These children of the interior window are called *body windows*. The client may create an arbitrary number of body windows and may arrange them arbitrarily. **Note:** Because the body windows are children of the interior window, they are clipped by the interior window. A client could, for example, create a body window that is very much taller than the interior window and accomplish scrolling simply by sliding the body window around inside the interior window (This is what the default **StarWindowShell** scrolling does; for more detail, see the section on scrolling).

The **StarWindowShell** interface provides a number of facilities for manipulating **StarWindowShells** and their various parts: creating and destroying a **StarWindowShell**; using body windows, commands and pop-up menus; client **TransitionProcs** (called whenever a **StarWindowShell** changes state--is opened or closed, for example); scrolling ; **AdjustProcs** and **LimitProcs**; and displaying and stacking (that is, open-within) **StarWindowShells**. The most commonly used facilities (creating a **StarWindowShell** and body windows) are described here and in the section on interface items. The less commonly used facilities are described in each subsection of the interface items.

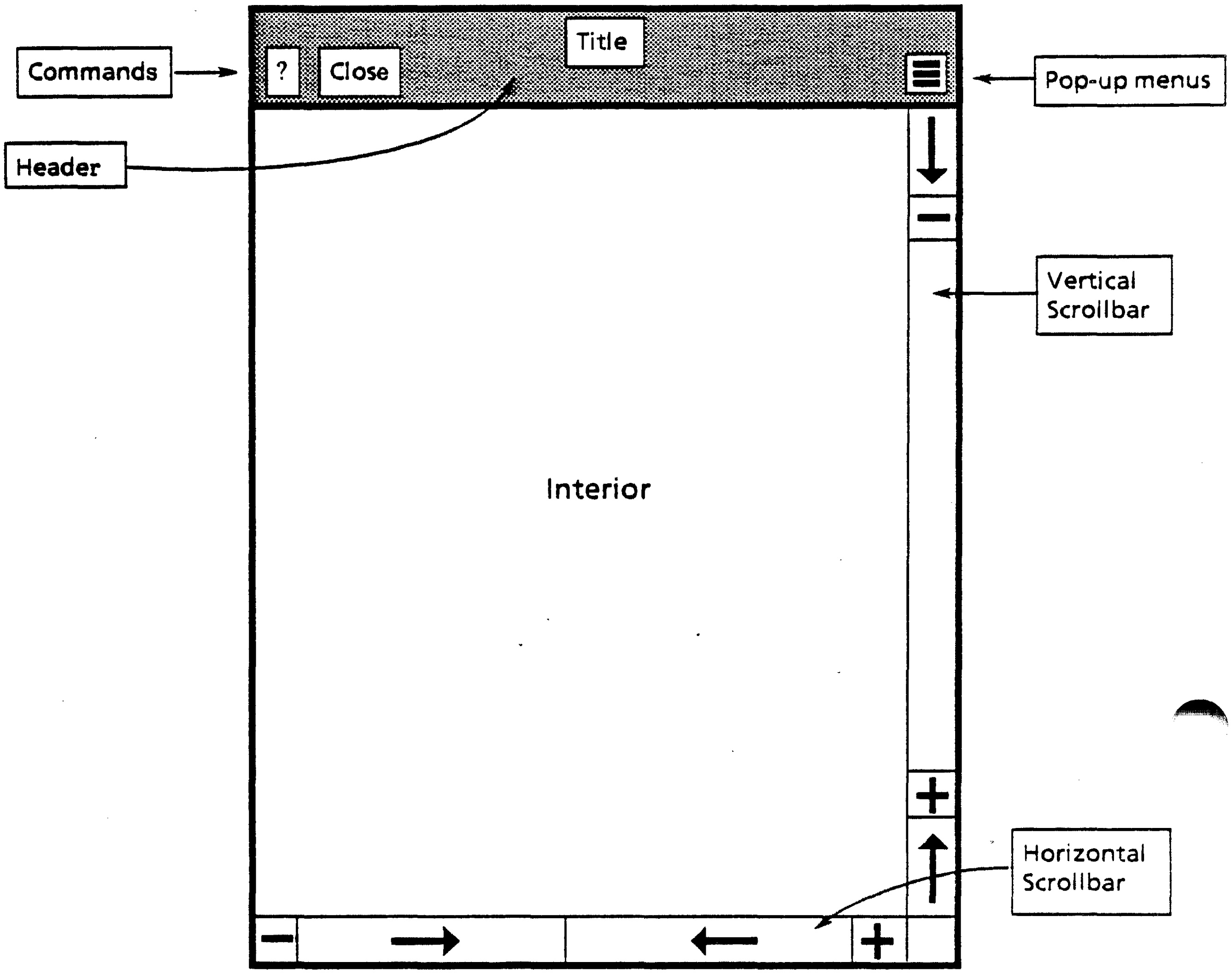


Figure 50.1 A Star Window Shell

50.1.2 Creating a StarWindowShell, Handles, etc.

A **StarWindowShell** is created by calling **StarWindowShell.Create**. There are no required parameters, but it is quite common to provide a **name** and a **transitionProc**. The **name** is displayed as the title in the **StarWindowShell** header. The **transitionProc** is called whenever the **StarWindowShell** is opened, destroyed, or "put to sleep," giving the client an opportunity to allocate and deallocate storage, open and close files, and so forth.

StarWindowShell.Create returns a **StarWindowShell.Handle**. A **StarWindowShell.Handle** is a **RECORD [Window.Handle]**. Thus any procedure that takes a **Window.Handle** also takes a **StarWindowShell.Handle**, but not the other way around. (The Mesa compiler automatically strips off the brackets and passes the **Window.Handle** if a **StarWindowShell.Handle** is passed). In particular, a context may be hung directly off a **StarWindowShell** (see the **Context**

interface). The **Handle** returned by **Create** is then used as the first parameter to most other calls to **StarWindowShell**.

The **StarWindowShell** returned by **Create** is not displayed on the screen (that is, it. is not inserted into the visible window tree). A **StarWindowShell** may be inserted into the window tree by calling **StarWindowShell.Push**. This is usually not done by the client but rather by some other part of ViewPoint, such as the desktop implementation. For example, when the user selects an icon and presses OPEN or PROPS, the application (actually the application's **Containee.GenericProc**) creates a **StarWindowShell** and returns it. The desktop implementation then displays the **StarWindowShell** by doing a **StarWindowShell.Push**.

50.1.3 Body Windows

Body windows are created by calling **StarWindowShell.CreateBody**. This returns a **Window.Handle**. The client can create an arbitrary number of body windows. Each body window is a child of the **StarWindowShell**'s interior window. The body windows may overlap or not. They can actually be in any arrangement the client finds useful. Some common arrangements of body windows are as follows:

- One very long body window.
  This is easy to scroll by simply sliding the body window, which is what the **StarWindowShell** default scolling does.

- One body window with **BodyWindowJustFits** = TRUE.
  This is one way to display an infinite amount of data, such as a Tajo-like editor. The client must keep track of what is currently in the window, use adjust procedures, do scrolling, and so forth. This is difficult to implement.

- Several body windows about the size of the interior, adjacent, non-overlapping.
  This is another way to display an infinite amount of data. The client lets **StarWindowShell** do default scrolling, which slides the body windows up or down and then calls the client to supply more body windows when it runs out. The client might put one page of text into each body window, supplying pages to **StarWindowShell** scrolling as needed.

- Several body windows smaller than the interior, adjacent, non-overlapping.
  This can be used to simulate subwindows.

Note: Body windows can themselves have child windows, and so on. A client might implement frames in a document editor by making each frame a child of a body window.

The eldest body window may be obtained by calling **StarWindowShell.GetBody**. All the body windows may be enumerated by calling **StarWindowShell.EnumerateBodiesInDecreasingY** or **StarWindowShell.EnumerateBodiesInIncreasingY**. To get the **StarWindowShell** from any body window, use **StarWindowShell.ShellFromChild**. Fine point: The client's body windows may not be the only child windows of the interior window, and the interior window may not be the only child of the **StarWindowShell** window. Therefore the client should never try to enumerate body windows by calling **Window.GetChild** and **Window.GetSibling** starting with the **StarWindowShell**, and the client should never try to get the **StarWindowShell** from a body window by calling **Window.GetParent**.

The client may provide a **repaintProc** and a **bodyNotifyProc** with each body window. The **repaintProc** is the display procedure that is called by the window implementation whenever part or all of the window needs to be displayed (see **Window.SetDisplayProc**). The **bodyNotifyProc** is a **TIP.NotifyProc** that is attached to the window along with the normal set of TIP tables and receives notifications for the window (see **TIP.SetNotifyProcAndTable**). **Note:** If the client is going to use some ViewPoint interface to turn the body window into a particular type of window (such as **FormWindow** or **ContainerWindow**), these procedures should not be supplied by the client, but rather are supplied by that interface.

A single body window can be set to fit into the interior window. Any time the **StarWindowShell's** size is changed, the body window's size is changed accordingly. (See **SetBodyWindowJustFits**.)

50.1.4 Commands and Menus

Every **StarWindowShell** can have commands and pop-up menus, as in Figure 50.1. Commands are actually individual menu items (**MenuData.ItemHandle**), where the **MenuData.ItemName** appears with a rounded corner box around it. When the user clicks over a command, the **MenuData.MenuProc** for that item is called. Commands are specified by calling **StarWindowShell.SetRegularCommands**, which takes a **MenuData.MenuHandle**. Each item in the menu is displayed as a command on the left side of the header.

A pop-up menu is an entire menu. The menu's title appears with a rounded corner box around it on the right side of the shell's header. When the user buttons down over the menu's title, a small window appears next to the poimter with one line for each menu item. When the user selects one of the items, that item's **MenuData.MenuProc** is called. Pop-up menus are specified by calling **StarWindowShell.AddPopupMenu**.

Facilities are also provided for specifying commands that should appear when a shell has other shells opened on top of or within it. (See the section on Push and Pop for a full discussion of the "open within" illusion, and the section on commands and menus for a full discussion of these extra commands.)

50.2 Interface Items

50.2.1 Create a StarWindowShell, etc.

```
Create: PROCEDURE [
   transitionProc: TransitionProc ← NIL,
   name:XString.Reader ← NIL,
   namePicture:XString.Character ← XChar.null,
   host: Handle ← NIL,
   type: ShellType ← regular,
   sleeps: BOOLEAN ← FALSE,
   considerShowingCoverSheet: BOOLEAN ← TRUE,
   currentlyShowingCoverSheet: BOOLEAN ← FALSE,
   pushersAreReadonly: BOOLEAN ← FALSE,
   readonly: BOOLEAN ← FALSE,
   scrollData: ScrollData ← vanillaScrollData,
   garbageCollectBodiesProc: PROCEDURE [Handle] ← NIL,
   isCloseLegalProc: IsCloseLegalProc ← NIL,
   bodyGravity: Window.Gravity ← nw,
zone: UNCOUNTED ZONE ← NIL]
RETURNS [Handle];
```

Create makes a **StarWindowShell** and returns a **Handle** to it. The **StarWindowShell** returned by **Create** is not displayed on the screen (is not inserted into the window tree). A **StarWindowShell** may be inserted into the window tree by calling **StarWindowShell.Push**. This is usually not done by the client but rather by some other part of ViewPoint, such as the desktop implementation. For example, when the user selects an icon and presses OPEN or PROPS, the application (actually the application's **Containee.GenericProc**) creates a **StarWindowShell** and returns it. The desktop implementation then displays the **StarWindowShell** by doing a **StarWindowShell.Push**.

transitionProc is a procedure that is called whenever the state of the shell is about to change. In particular, it is called just before the shell is destroyed. The client uses a **transitionProc** to free any data structures that may have been allocated and associated with the shell. **TransitionProcs** are discussed in later in this chapter.

name appears as the title in the header of the **StarWindowShell**.

namePicture appears just before the title in the header. This character is usually a small icon picture created by **SimpleTextFont.AddClientDefinedCharacter**.

host is a **StarWindowShell** that this shell is logically attached to. The **host** shell is not destroyed while this shell is open. This is typically used by property sheets to indicate the shell that the property sheet is displaying properties of. If **host** is NIL, closing this shell does not depend on any other shell.

type is the type of the shell. Shell placement algorithms may be affected by the type. For example, **regular** shells will not overlap when displayed with Star-style window management, while **psheet** shells may overlap other shells.

sleeps indicates whether this shell can go into the **sleeping** **StarWindowShell.State**. If it is FALSE, we assume that the client software does not take advantage of the possibilities of the **sleeping State** (by remembering data from open to open). This argument is used with the client's **transitionProc**, discussed later in this chapter.

considerShowingCoverSheet and **currentlyShowingCoverSheet** indicate whether the shell should ever possess a cover sheet and, if so, whether the cover sheet should be visible. What appears in any cover sheet is governed by a cover sheet implementation. See the section on Errors.

The two **readonly** arguments define whether this shell is uneditable and whether all shells pushed onto this one should be uneditable. **readonlyness** is really up to client interpretation. This information is simply maintained for client convenience. If a shell below this one in a push stack has **pushersAreReadonly** set TRUE, then the implementation forces **readonly** to TRUE.

scrollData indicates whether vertical or horizontal scrollbars should appear and allows the client to supply procedures to be called for various user scrolling actions. (See the section on scrolling for full details.) The default will cause vertical scrollbars to appear, but not horizontal. The default scrolling procedures simply slide body windows up or down, left or right, as appropriate.

garbageCollectBodiesProc is called when a scroll action causes a body window to be placed completely outside the shell's interior window. The call thus allows the client an opportunity to garbage-collect the body window and associated data structures. (See the section on scrolling.)

isCloseLegalProc is called when the user attempts to close the **StarWindowShell** or when a client calls **StandardClose**, **StandardCloseAll**, or **StandardCloseEverything**. This allows the client to veto the user's attempt to close the window. If the **isCloseLegalProc** returns TRUE, the shell is closed; if the **isCloseLegalProc** returns FALSE, the shell is not closed. The **isCloseLegalProc** is also a convenient way for the client to get control when the window is being closed.

bodyGravity argument indicates what value for **gravity** should be used when the implementation adjusts the size of a body window.

zone defines the storage area from which all the shell data structures are allocated If **zone** = NIL, **StarWindowShell** creates a zone which will be destroyed when the shell is destroyed. However, if the client supplies a zone which should be destroyed when the Shell is destroyed, then the client should call **SetDestroyZoneProc** (see below). Fine point: The **Window.Objects** themselves are not allocated out of the client's zone. If the client allocates child windows using a zone (**Window.Create** or **Window.New** with non-NIL zone), these child windows must be removed from the shell before it is destroyed. When the shell's **TransitionProc** is called with a state of dead, the client should remove those windows.

```
CreateWithBitmapUnderOption: PROCEDURE [
   transitionProc: TransitionProc ← NIL,
   name:XString.Reader ← NIL,
   namePicture:XString.Character ← XChar.null,
   host: Handle ← NIL,
   type: ShellType ← regular,
   sleeps: BOOLEAN ← FALSE,
   considerShowingCoverSheet: BOOLEAN ← TRUE,
   currentlyShowingCoverSheet: BOOLEAN ← FALSE,
   pushersAreReadonly: BOOLEAN ← FALSE,
   readonly: BOOLEAN ← FALSE,
   scrollData: ScrollData ← vanillaScrollData,
   garbageCollectBodiesProc: PROCEDURE [Handle] ← NIL,
   isCloseLegalProc: IsCloseKegalProc ← NIL,
   bodyGravity: Window.Gravity ← nw,
   bitmapOption: BitmapUnderOption ← noBitmapUnder,
   zone: UNCOUNTED ZONE ← NIL ]
   RETURNS [Handle];
```

```
BitmapUnderOption: TYPE = {noBitmapUnder, maybeBitmapUnder, bitmapUnder};
```

CreateWithBitmapUnderOption is identical to **Create** except for the additional parameter **bitmapOption**. If **bitmapOption** is **noBitmapUnder**, no attempt is made to make the shell a bitmap-under. If **bitmapOption** is **bitmapUnder**, the shell will be a bitmap-under window if enough storage space is available. If **bitmapOption** is **maybeBitmapUnder**, StarWindowShell will use **ShellType** to decide. Currently only shells of type **psheet** will be bitmap-under windows if **bitmapOption** is **maybeBitmapUnder**. Note that StarWindowShell will manage the space used for the shell's bitmap-under. Fine Point: **CreateWithBitmapUnderOption** and **BitmapUnderOption** are currently exported through **StarWindowShellExtra5.**

Handle: TYPE = RECORD [Window.Handle];

Create and **CreateWithBitmapOption** returns a **Handle**. Any procedure that takes a **Window.Handle** also takes a **StarWindowShell.Handle**, but not the other way around. (The Mesa compiler automatically strips off the brackets and passes the **Window.Handle** if a **StarWindowShell.Handle** is passed). In particular, a **Context** may be hung directly off a **StarWindowShell** (see the **Context** interface). The **Handle** returned by **Create** is then used as the first parameter to most other calls to **StarWindowShell**.

nullHandle: Handle = [NIL];

nullHandle is provided as a convenience.

ManagerFromShell: PROCEDURE [sws: Handle]
RETURNS [swmanager: Window.Handle];

Given a shell (**sws**) return its **swmanager**. The **swmanager** is the descendant window of the shell which "just fits" in the shell's interior. It may be used in conjunction with the subwindow interfaces to divide the shell into subwindows. See the **Subwindow Overview** Chapter for more information concerning the use of the subwindow package.

IsCloseLegalProc: TYPE = PROCEDURE [sws: Handle,
closeAll: BOOLEAN ← FALSE] RETURNS [BOOLEAN];

closeAll indicates whether the user selected Close or CloseAll.

Destroy: PROCEDURE [sws: Handle];

Destroys the **StarWindowShell** and associated data. First, the client's **transitionProc** is called with **state** = **dead**. Next, all the shell data is freed. Finally, if the **zone** was supplied by the client then any **DestroyZoneProc** registered by **SetDestroyZoneProc** is called. Otherwise, the**StarWindowShell** created **zone** is destroyed. May raise **Error [notASWS]**.

DestroyZoneProc: TYPE = PROCEDURE [z: UNCOUNTED ZONE, clientData: LONG POINTER ← NIL];

SetDestroyZoneProc: PROCEDURE [
sws: Handle, proc: DestroyZoneProc ← DefaultDestroyZoneProc,
clientData: LONG POINTER ← NIL] RETURNS [oldProc: DestroyZoneProc, oldClientData: LONG POINTER];

A **DestroyZoneProc** is called at the end of the process of destroying a shell, allowing the client to destroy a zone that was created before the shell was created. That is, if the client supplies a **zone** parameter to **Create**, then destroying that zone should be done in a **DestroyZoneProc**. **SetDestroyZoneProc** associates a **proc** with the **sws**. **oldProc** and **oldClientData** are the previous values or NIL. See Usage/Examples.

DefaultDestroyZoneProc: DestroyZoneProc;

DefaultDestroyZoneProc calls **Heap.Delete**, so it may be used by clients that use the **Heap** package for the **zone** parameter to **Create**.

GetDestroyZoneProc: PROCEDURE [sws: StarWindowShell.Handle]
RETURNS [oldProc: DestroyZoneProc, oldClientData: LONG POINTER];

GetDestroyZoneProc returns the current settings for DestroyZoneProc and clientData.
Fine point: **DestroyZoneProc, SetDestroyZoneProc, GetDestroyZoneProc** and DefaultDestroyZoneProc are defined in **StarWindowShellExtra5.mesa**

ShellType: TYPE = {regular(0), keyboard, psheet, attention, static, last(15)};

ShellType influences how a shell behaves in several regards. **regular** shells have a ? command, a Close command, and a Close All command if opened on top of another shell. With Star-like overall screen management, **regular** shells do not overlap; they change size whenever a window is opened or closed. **psheet** shells do not have any **StarWindowShell**-supplied commands and freely overlap other shells. The **PropertySheet** interface uses **psheet** shells to create property sheets. **static** shells are exempted from any overall screen management; for example, a **static** shell is not shrunk to make room for a **regular** shell when the overall screen management is Star-like. Some clients may find this useful. Most clients do not use **keyboard, psheet,** or **attention** types.

StandardClose: PROCEDURE [sws: Handle] RETURNS [Handle];

StandardCloseAll: PROCEDURE [sws: Handle] RETURNS [Handle];

StandardClose and **StandardCloseAll** provide procedural access to the Close and Close All commands that are placed in a shell's header automatically by **StarWindowShell**. These procedures call the client's **isCloseLegalProc** and **transitionProc**, just as if the user had selected the command. If **StandardClose** or **StandardCloseAll** is not successful, the return parameter is the shell that did not close; otherwise, the return parameter is **NIL**. May raise **Error [notASWS]**.

StandardCloseEverything: PROCEDURE RETURNS [notClosed: Handle];

StandardCloseEverything closes all open **StarWindowShells**. Logoff uses this procedure. **notClosed** is the first window that could not be closed because its **IsCloseLegalProc** returned FALSE. All windows that can be closed will be. If **notClosed** is NIL, then all windows are closed.

NewStandardCloseEverything: PUBLIC PROCEDURE
RETURNS [numberLeftOpen: CARDINAL ← 0, lastNotClosed: Handle ← nullHandle];

This procedure is the same as **StandardCloseEverything** except that it also returns the number of shells that vetoed close. **NewStandardCloseEverything** is defined in StarWindowShellExtra.mesa.

SetPreferredDims: PROCEDURE [sws: Handle, dims: Window.Dims];

SetPreferredPlace: PROCEDURE [sws: Handle, place: Wndow.Place];

SetPreferredDims and **SetPreferredPlace** provide a suggestion as to the desired size and location of the shell. Depending on the overall screen management in effect at the time the shell is displayed, these preferred values may be ignored. May raise **Error [notASWS]**.

GetPreferredDims: PROCEDURE [sws: Handle] RETURNS [box: Window.Dims];

GetPreferredPlace: PROCEDURE [sws: Handle] RETURNS [box: Wndow.Place];

GetPreferredDims and **GetPreferredPlace** return the current preferred dims and place of **sws**. May raise **Error [notASWS]**. These are defined in StarWindowShellExtra4.mesa.

SetPreferredInteriorDims: PROCEDURE [sws: Handle, dims: Window.Dims];

SetPreferredInteriorDims makes the shell just big enough to fit around **dims**. This means the interior window will be of size **dims**. **SetPreferredInteriorDims** is defined in StarWindowShellExtra2.mesa.

[fine point: there is a gridding procedure used when sizing and placing shells on the desktop pattern that may cause the shell to be resized or moved ±2 bits.]

50.2.1.1 IsCloseLegalProc

The client may supply an **isCloseLegalProc** when a **StarWindowShell** is created or later by calling **SetIsCloseLegalProc**. This client procedure is called when the user attempts to close the **StarWindowShell** or when a client calls **StandardClose**, **StandardCloseAll**, or **StandardCloseEverything**. This allows the client to veto the user's attempt to close the window. If the **isCloseLegalProc** returns TRUE, the shell is closed; if the **isCloseLegalProc** returns FALSE, the shell is not closed. The **isCloseLegalProc** is also a convenient way for the client to get control when the window is being closed.

IsCloseLegal: PROCEDURE [sws: Handle, closeAll: BOOLEAN] RETURNS [BOOLEAN];

IsCloseLegal calls the client's **isCloseLegalProc** and returns the value returned from that call. If there is no **isCloseLegalProc**, **IsCloseLegal** returns TRUE. May raise **Error [notASWS]**.

IsCloseLegalProcReturnsFalse: IsCloseLegalProc;

```
GetIsCloseLegalProc: PROCEDURE [sws: Handle]
   RETURNS [ IsCloseLegalProc ];
```

GetIsCloseLegalProc returns the current **isCloseLegalProc** associated with **sws**. May raise **Error [notASWS]**.

```
SetIsCloseLegalProc: PROCEDURE [
   sws: Handle,
   proc: IsCloseLegalProc ];
```

SetIsCloseLegalProc sets the **isCloseLegalProc** for **sws**. May raise **Error [notASWS]**.

Note: **IsCloseLegalProc: TYPE = PROCEDURE [...** should be in this interface and will be added in the next release.

50.2.1.2 Miscellaneous Get and Set Procedures

Several procedures that set and return values logically associated with a **StarWindowShell** are provided.

GetContainee: PROCEDURE [sws: Handle] RETURNS [Containee.Data];

GetHost: PROCEDURE [sws: Handle] RETURNS [Handle];

GetName: PROCEDURE [sws: Handle, callBack: PROCEDURE [name: XString.Reader]];

GetReadonly: PROCEDURE [sws:Handle] RETURNS [BOOLEAN] ;

GetType: PROCEDURE [sws:Handle]RETURNS[ShellType];

These procedures return the obvious value associated with **sws**. May raise **Error [notASWS]**.

GetZone: PROCEDURE [sws: Handle] RETURNS [UNCOUNTED ZONE];

When a **StarWindowShell** is created the client may specify a zone, or the **StarWindowShell** implementation may create a zone which will contain all shell-related data items. The client can use this zone which is returned by **GetZone**. If the zone was created by the **StarWindowShell** implementation then it will be completely garbage-collected when the shell is destroyed. However, if the zone was supplied by the client then the fate of the zone is up to the client. See also the **DestroyZoneProc** discussion in section 50.2.1. May raise **Error [notASWS]**.

HaveDisplayedParasite: PROCEDURE [sws: Handle] RETURNS [BOOLEAN];

HaveDisplayedParasite returns TRUE if a shell is displayed that has this shell (**sws**) as its host. (See **host** under **StarWindowShell.Create**.) For example, if a property sheet that was created with **host** = **sws** is currently displayed, then **HaveDisplayedParasite [sws]** returns TRUE. May raise **Error [notASWS]**.

SetContainee: PROCEDURE [sws: Handle, file: Containee.DataHandle];

SetHost: PROCEDURE [sws, host: Handle] RETURNS [old: Handle];

SetName: PROCEDURE [sws: Handle, name: XString.Reader];

SetNamePicture: PROCEDURE [sws: Handle, picture: XString.Character];

SetReadOnly: PROCEDURE [sws: Handle, yes: BOOLEAN];

SetSleeps: PROCEDURE [sws: StarWindowShell.Handle, sleeps: BOOLEAN]
RETURNS [old: BOOLEAN];

sleeps = TRUE means the shell can be put to sleep. It is the same as the **sleeps** parameter to **Create**. Fine point: This procedure is currently exported through **StarWindowShellExtra**.

These procedures set the obvious value associated with **sws**. May raise **Error [notASWS]**.

50.2.2 Body Windows

A **StarWindowShell** is a window (see **Window** interface) that is a child of the desktop window. A **StarWindowShell** has an interior window that is a child of the **StarWindowShelland** is exactly the size of the available window space in the shell (that is, the **StarWindowShell** minus its borders and header and scrollbars). The interior window may have child windows created by the client. These children of the interior window are called *body windows*. The client may create an arbitrary number of body windows and arrange them in an arbitrary fashion. **Note:** Since the body windows are children of the interior window, they are clipped by the interior window. A client could, for example, create a body window that is very much taller than the interior window and accomplish scrolling by simply sliding the body window around inside the interior window. (This is actually what the default **StarWindowShell** scrolling does; for more detail, see the section on scrolling).

Body windows are created by calling **StarWindowShell.CreateBody**. This returns a **Window.Handle**. The client can create an arbitrary number of body windows. Each body window is a child of the **StarWindowShell**'s interior window. The body windows may overlap or not. They can be in any arrangement the client finds useful. Some common arrangements of body windows are as follows:

- One very long body window.
  This is easy to scroll by simply sliding the body window, which is what the **StarWindowShell** default scrolling does.

- One body window with **BodyWindowJustFits = TRUE**.
  This is one way to display an infinite amount of data, such as a Tajo-like editor. The client must keep track of what is currently in the window, use adjust procs, do scrolling, and so forth. This is difficult to implement.

- Several body windows about the size of the interior, adjacent, non-overlapping.
  This is another way to display an infinite amount of data. The client lets **StarWindowShell** do default scrolling, which slides the body windows up or down and then calls the client to supply more body windows when it runs out. The client might put one page of text into each body window, supplying pages to **StarWindowShell** scrolling as needed.

- Several body windows smaller than the interior, adjacent, non-overlapping.
  This can be used to simulate subwindows.

Note: Body windows can themselves have child windows, and so on. A client might implement frames in a document editor by making each frame a child of a body window.

```
CreateBody: PROCEDURE [
   sws: Handle,
   repaintProc: PROCEDURE [Window.Handle] ← NIL,
   bodyNotifyProc: TIP.NotifyProc ← NIL,
   box: Window.Box ← [[0,0],[0,29999]] ] RETURNS [Window.Handle];
```

CreateBody creates a body window that is a child of the interior window of **sws**. **repaintProc** is the display procedure that is called by the window implementation whenever part or all of the body window needs to be displayed (see **Window.SetDisplayProc**).

bodyNotifyProc is a TIP.NotifyProc that is attached to the window along with the normal set of TIP tables and receives notifications for the window (see TIP.SetNotifyProcAndTable and TIPStar.NormalTable). **Note:** If the client is going to use some ViewPoint interface to turn the body window into a particular type of window (such as FormWindow or ContainerWindow), these procedures should not be supplied by the client but are supplied by that interface. **box** indicates the size and location of the body window within the shell's interior window. If **box.dims.w** and/or **box.dims.h** is zero, the body window takes on the **dims.w** and/or **dims.h** of the shell's interior window. May raise Error [notASWS].

```
DestroyBody: PROCEDURE [body: Window.Handle];
```

DestroyBody destroys **body** and any **Context** data associated with it. May raise Error [notASWS].

```
GetBody: PROCEDURE [sws: Handle] RETURNS [ Window.Handle];
```

GetBody returns the eldest body window of **sws**. The client's body windows may not be the only child windows of the interior window, and the interior window may not be the only child of the StarWindowShell window. Therefore the client should never try to enumerate body windows by calling Window.GetChild and Window.GetSibling starting with the StarWindowShell. The EnumerateBodiesXXX procedures should be used instead. May raise Error [notASWS].

```
ShellFromChild: PROCEDURE [child: Window.Handle] RETURNS [ Handle];
```

ShellFromChild returns the shell given any body window or any descendant window of the shell. The client's body windows may not be the only child windows of the interior window, and the interior window may not be the only child of the StarWindowShell window. Therefore the client should never try to get the StarWindowShell from a body window by calling Window.GetParent. May raise Error [notASWS].

```
EnumerateBodiesInIncreasingY: PROCEDURE [
  sws: Handle, proc: BodyEnumProc] RETURNS [Window.Handle ← NIL];

EnumerateBodiesInDecreasingY: PROCEDURE [
  sws: Handle, proc: BodyEnumProc] RETURNS [Window.Handle ← NIL];

BodyEnumProc: TYPE = PROCEDURE [victim: Window.Handle]
  RETURNS [stop: BOOLEAN ← FALSE];
```

The **EnumerateBodiesIn...** procedures enumerate all the body windows of **sws**, calling **proc** for each body window until **proc** returns **stop** = TRUE. **EnumerateBodiesInIncreasingY** enumerates the body windows in increasing order of Window.GetBox [body].place.y, and **EnumerateBodiesInDecreasingY** enumerates the body windows in decreasing order of Window.GetBox [body].place.y. Each procedure returns the last body window enumerated or NIL if all body windows were enumerated. These procedures are especially handy for clients that do their own scrolling. To minimize repainting when scrolling a set of body windows upward, it is important to move the upper ones first, and vice versa. May raise Error [notASWS].

```
GetBodyWindowJustFits: PROCEDURE [sws: Handle] RETURNS [ BOOLEAN];
```

SetBodyWindowJustFits: PROCEDURE [sws: Handle, yes: BOOLEAN];

Some clients may wish to have a single body window that is automatically resized by the **StarWindowShell** implementation to just fit within the interior of the shell. Such a body window is said to have **BodyWindowJustFits** = TRUE. If **BodyWindowJustFits** = FALSE (the default for **CreateBody**), **StarWindowShell** leaves the body window's dimensions alone, even though the body window may stick out or not fill the shell. **GetBodyWindowJustFits** and **SetBodyWindowJustFits** allow the client to determine and set this just-fits behavior for a single body window. Setting just-fits when there is more than one body window yields undefined results. May raise **Error [notASWS]**.

GetAvailableBodyWindowDims: PROCEDURE [sws: Handle]
  RETURNS [Window.Dims];

GetAvailableBodyWindowDims returns the current dimensions of the interior window of **sws**. May raise **Error [notASWS]**.

IsBodyWindowOutOfInterior: PROCEDURE [body: Window.Handle]
  RETURNS [BOOLEAN];

IsBodyWindowOutOfInterior returns TRUE if all of **body** is sticking out of the interior window of its shell. i.e. none of body is visible in its parent. May raise **Error [notASWS]**.

InstallBody: PROCEDURE [sws: Handle, body: Window.Handle];

InstallBody installs a previously created window into a **StarWindowShell**, thus making the window a body window. Most clients do not need to use this procedure. May raise **Error [notASWS]**.

DestallBody: PROCEDURE [body: Window.Handle];

DestallBody removes **body** from its **StarWindowShell**. Most clients do not need to use this procedure. May raise **Error [notASWS]**.

50.2.3 Commands and Menus

Every **StarWindowShell** can have commands and pop-up menus, as in Figure 50.1. Commands are actually individual menu items (**MenuData.ItemHandle**), in which the **MenuData.ItemName** appears with a rounded corner box around it. When the user clicks over a command, the **MenuData.MenuProc** for that item is called. Commands are specified by calling **StarWindowShell.SetRegularCommands**, which takes a **MenuData.MenuHandle**. Each item in the menu is displayed as a command on the left side of the header.

A pop-up menu is an entire menu. The menu's title appears with a rounded corner box around it on the right side of the shell's header. When the user buttons down over the menu's title, a small window appears next to the pointer with one line for each menu item. When the user selects one of the items, that item's **MenuData.MenuProc** is called. Pop-up menus are specified by calling **StarWindowShell.AddPopupMenu**.

The Window.Handle that is passed to the MenuData.MenuProc for a command or pop-up menu item is the Window.Handle for the StarWindowShell that the command or pop-up menu is currently displayed in.

StarWindowShells that are of type regular (see StarWindowShell.ShellType) always have system commands leftmost in the header. When a shell is directly on the desktop, the system command is Close. When a shell is opened within another, the system commands are Close and Close All.

Note: Commands may be added to and removed from a StarWindowShell by using MenuData.AddItem,and so forth.

The implementation automatically overflows the rightmost commands into an overflow pop-up menu when all of them will not fit in the header. If all the pop-up menus will not fit in the header, the leftmost items are overflowed into the rightmost pop-up menu. The rightmost pop-up menu is always guaranteed to be displayed, because shells are not allowed to be so small that no pop-up menu will fit.

```
SetRegularCommands: PROCEDURE [
   sws: Handle, commands: MenuData.MenuHandle] ;
```

SetRegularCommands associates commands with sws. May raise Error [notASWS].

```
GetRegularCommands: PROCEDURE [sws: Handle]
   RETURNS [MenuData.MenuHandle];
```

GetRegularCommands returns the regular commands associated with sws. May raise Error [notASWS].

```
AddPopupMenu: PROCEDURE [
   sws: Handle, menu: MenuData.MenuHandle] ;
```

```
AddPopupMenuX: PROCEDURE [
   sws: StarWindowShell.Handle, menu: MenuData.MenuHandle, repaint: BOOLEAN];
```

AddPopupMenu and AddPopupMenuX add menu to the available pop-up menus in sws. The title of menu is displayed in the StarWindowShell header with the small pop-up menu symbol (▪) just to the left of it, enclosed in a rounded corner box. Note: Any arbitrary symbol (less than the height of the system font) can be part of the title by using SimpleTextFont.AddClientDefinedCharacter. AddPopupMenu causes an immediate redisplay of the header. AddPopupMenuX only repaints if repaint = TRUE. May raise Error [notASWS].

```
SubtractPopupMenu: PROCEDURE [
   sws: Handle, menu: MenuData.MenuHandle] ;
```

```
SubtractPopupMenuX: PROCEDURE [
   sws: StarWindowShell.Handle, menu: MenuData.MenuHandle, repaint: BOOLEAN];
```

SubtractPopupMenu and SubtractPopupMenuX remove menu from sws. SubtractPopupMenu causes an immediate redisplay of the header. SubtractPopupMenuX only repaints if repaint = TRUE. May raise Error [notASWS].

EnumeratePopupMenus: PROCEDURE [sws: Handle, proc: MenuEnumProc];

Enumerates the pop-up menus associated with the shell.

EnumerateAllMenus: PROCEDURE [sws: Handle, proc: MenuEnumProc];

Enumerates every menu associated with the shell. This includes pop-ups, regular commands, **topPushee** commands from the shell underneath, etc. Fine point: This procedure is currently exported through **StarWindowShellExtra**.

EnumerateDisplayedMenus: PROCEDURE [sws: Handle, proc: MenuEnumProc];

Enumerates every menu visible in the shell. This includes pop-ups, regular commands, **topPushee** commands from the shell underneath, etc. Fine point: This procedure is currently exported through **StarWindowShellExtra6**.

```
MenuEnumProc: TYPE = PROCEDURE [menu: MenuData.MenuHandle]
  RETURNS [stop: BOOLEAN ← FALSE];
```

50.2.3.1 Pushee Commands

Facilities are also provided for specifying commands that should appear when a shell has had other shells opened on top of or within it. These facilities are useful only to a client that implements some type of open-within capability, such as folders and file drawers. (See the section on commands and menus for a full discussion of the "open within" illusion.) These extra commands come in three sets: the set that should be displayed when this shell is just below the top of the install stack, the set that should be displayed when this shell is anywhere in the install stack, and the set that should be displayed if this shell is at the bottom of an install stack. These are the so-called **TopPushee**, **MiddlePushee**, and **BottomPushee** commands.

Figure 50.2 depicts how these pushee commands, if supplied, will affect the commands visible in a given shell's header. In Figure 50.2, Shell B is **Pushed** on top of Shell A and Shell C is **Pushed** on top of Shell B. If Shell A is the only shell displayed, Shell A's system and regular commands appear in the shell's header. With Shell B **Pushed** on top of Shell A, Shell B's system and regular commands appear as well as Shell A's bottom pushee, middle pushee, and top pushee commands. This is because Shell A is on the bottom, in the middle, and just below the top of the stack of shells. With Shell C **Pushed** on top of Shell B, Shell A's bottom pushee and middle pushee commands appear, but not Shell A's top pushee commands. Shell B's top pushee and middle pushee commands appear, but not its bottom pushee commands.

Caution: The **Window.Handle** passed to the **MenuData.MenuProc** for any pushee command is the **Window.Handle** of the **StarWindowShell** that the command is currently displayed in, not the **StarWindowShell** that the command was originally associated with. If the client wants to be able to recover the **StarWindowShell** that the command was originally associated with, it may be saved as the **MenuData.ItemData**.

```
SetBottomPusheeCommands: PROCEDURE [
  sws: Handle, commands: MenuData.MenuHandle] ;
```

```
SetMiddlePusheeCommands: PROCEDURE [
   sws: Handle, commands: MenuData.MenuHandle] ;

SetTopPusheeCommands: PROCEDURE [
   sws: Handle, commands: MenuData.MenuHandle] ;

GetPusheeCommands: PROCEDURE [sws: Handle]
   RETURNS [bottom, middle, top: MenuData.MenuHandle];
```

May raise Error [notASWS].

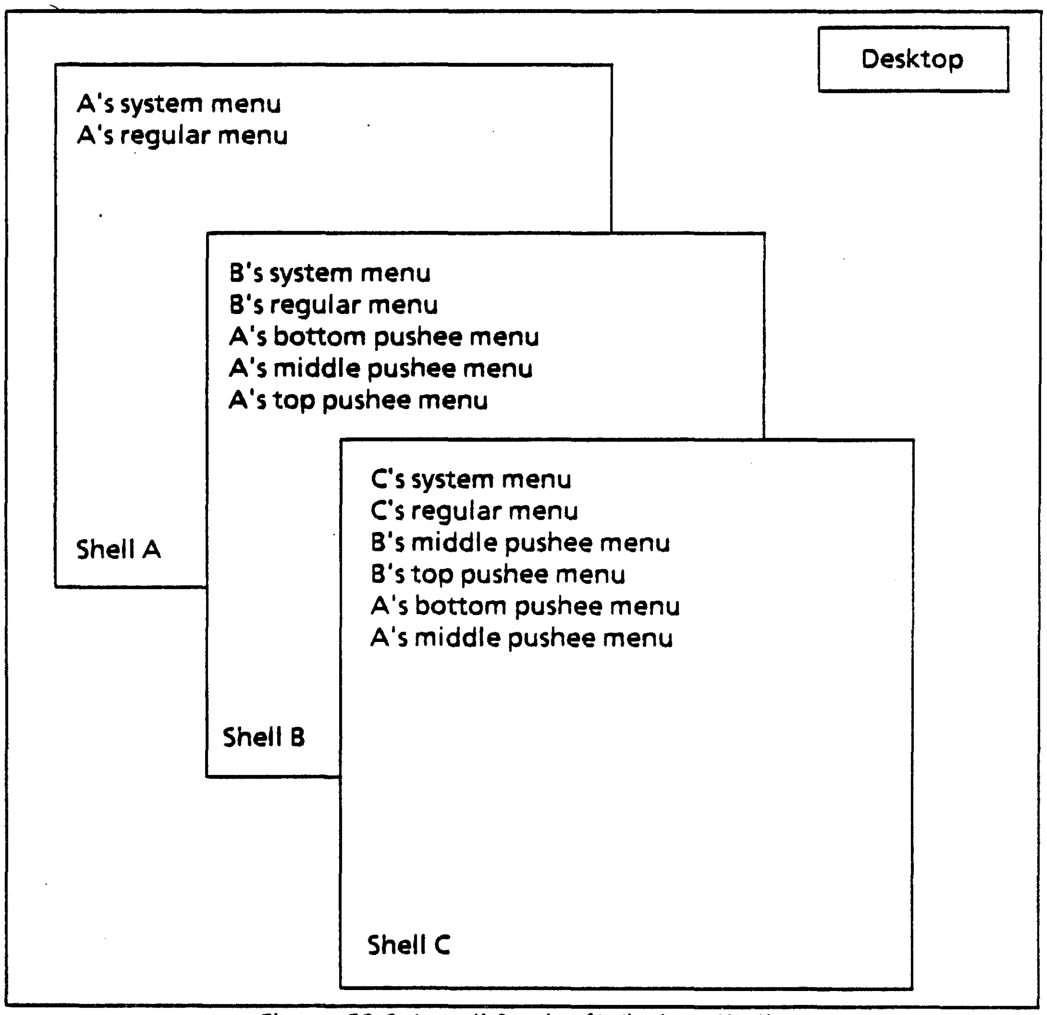


Figure 50-2. Install Stack of Window Shells

50.2.4 TransitionProcs

A **StarWindowShell** is always in one of three states: **awake**, **sleeping**, or **dead**. The **awake** state indicates that the shell is currently displayed. The **sleeping** state indicates that the shell still exists but is not being displayed and therefore resources associated with the display state should be freed. The **dead** state indicates the shell is about to be destroyed and therefore all resources associated with it should be freed.

Every **StarWindowShell** can have a client-supplied **TransitionProc** associated with it. This **TransitionProc** is called whenever the shell's state changes. The client may then do whatever is necessary in terms of allocating or freeing resources.

The client may call StarWindowShell.Create [... sleeps: FALSE ...] to indicate that the application does nothing interesting with the **sleeping** state. This may cause routines that would otherwise put the shell in **sleeping** state (say on a close, where it might be quickly reopened) to put it in **dead** state instead.

State: TYPE = {awake(0), sleeping, dead, last(7)};

TransitionProc: TYPE = PROCEDURE [sws: Handle, state: State];

The **TransitionProc** is a client-supplied procedure responsible for allocating or deallocating client data structures when **sws**'s state changes. **state** is the new state of **sws**.

When the **TransitionProc** is called with **state** = **dead**, the **zone** associated with the shell must not be destroyed yet. If the **zone** was supplied by the client and should be destroyed, this must be done in a **DestroyZoneProc** which will be called later during the **Destroy** process. See also the **DestroyZoneProc** discussion in section 50.2.1 and the GetZone discussion in 50.2.1.2.

GetSleeps: PROCEDURE[sws: Handle] RETURNS [BOOLEAN];

GetSleeps returns the value of the **sleeps** parameter that was passed to **Create** when **sws** was created. If TRUE, then the application responsible for this shell can deal with the **sleeping** state. May raise **Error [notASWS]**.

GetState: PROCEDURE [sws: Handle] RETURNS [State] ;

GetState returns the current state of **sws**. May raise **Error [notASWS]**.

GetTransitionProc: PROCEDURE [sws:Handle] RETURNS [TransitionProc];

GetTransitionProc returns the current **TransitionProc** associated with **sws**. May raise **Error [notASWS]**.

```
SetTransitionProc: PROCEDURE [sws: Handle, new: TransitionProc]
   RETURNS [ old: TransitionProc];
```

SetTransitionProc sets the current **TransitionProc** for **sws** and returns the old one. May raise **Error [notASWS]**.

SetState: PROCEDURE [sws: Handle, state: State] ;

SetState sets the state of **sws** and calls the client's **TransitionProc** as appropriate. Most clients will not call this procedure. May raise **Error [notASWS]**.

SleepOrDestroy: PROCEDURE [Handle] RETURNS[Handle];

SleepOrDestroy either puts the shell in the **sleeping** state or destroys the shell, depending on the value of the **sleeps** **BOOLEAN** that was passed to **Create** when the shell was created. If the shell has the ability to sleep (**sleep** = **TRUE**), then the shell is put into the **sleeping** state; otherwise, the shell is destroyed. The same shell is returned if the shell was put in the sleeping state. This returned **Handle** should be kept somewhere for later use; otherwise, the shell will be lost. A **NIL** handle is returned if the shell was destroyed. This procedure might be used by the desktop implementation when a shell is closed. May raise **Error [notASWS]**.

50.2.5 Scrolling

Only part of an object is usually visible to the user at any one moment in the interior of a **StarWindowShell**. The user can request to see more of the object by scrolling the contents up or down inside the shell. The user can perform three kinds of scrolling by using the scrollbars pictured in Figure 50.1. (1) He can move the contents a little at a time by pointing at the arrows (up, down, left, right) in the scrollbars. (2) He can move the contents a page or screenful at a time by pointing at the plus (+) and minus (-) signs. (3)He can jump to any arbitrary place within the entire extent of the object being viewed by pointing in the blank part of the vertical scrollbar (this latter operation is called *thumbing*).

StarWindowShell provides various levels of support to a client for performing these scrolling operations. The client can allow **StarWindowShell** to do all the scrolling functions, the client can do some of them and leave the rest to **StarWindowShell**, or the client can do all scrolling operations. Much of this decision will be based on how the client chooses to arrange body windows within the shell (see the section on body windows above and more discussion below). First, we will describe the various types of scrolling and scrolling procedures that a client can supply; then we will describe the default scrolling procedures provided by **StarWindowShell**.

In the simplest (for the client) case, one body window contains the entire extent of the object being viewed. **StarWindowShell** can handle all scrolling in this case. The client simply defaults the **scrollData** parameter in the call to **StarWindowShell.Create**. When the user points at an arrow, **StarWindowShell** moves the body window a small amount. When the user point at plus or minus, **StarWindowShell** moves the body window by one interior window's height. When the user thumbs, **StarWindowShell** will move the body window to an appropriate place based on its overall height.

In a slightly more complex case, body windows are butted up against one another. When the user points at an arrow, **StarWindowShell** moves all the body windows a small amount. When the user point at plus or minus, **StarWindowShell** moves all the body windows by one interior window's height. When the user thumbs, **StarWindowShell** moves all the body windows to an appropriate place based on the combined overall height of the body windows. However, in this case the client often does not have the entire extent of the object displayed in these body windows but rather wants to tack new body windows on each end as these body windows are scrolled off. The client can do this by providing a **MoreScrollProc** for the shell. **StarWindowShell** calls the client's **MoreScrollProc** whenever it runs out of body windows.

In the most complex case, the client has a single body window that "just fits" (see **SetBodyWindowJustFits** in the section on body windows), and only part of the entire object is displayed at any one time. The client must provide *all* the scrolling functions for this case. This means providing an **ArrowScrollProc** (to handle the user's pointing at the arrows, plus, and minus) and a **ThumbScrollProc** (to handle the user's thumbing).

Of course, the client may provide its own scrolling procedures for any of the above cases, even the simple one, to override the type of scrolling that **StarWindowShell** provides.

```
ScrollData: TYPE = RECORD [
   displayHorizontal: BOOLEAN ← FALSE,
   displayVertical: BOOLEAN ← FALSE,
   arrowScroll: ArrowScrollProc ← NIL,
   thumbScroll: ThumbScrollProc ← NIL,
   moreScroll: MoreScrolllProc ← NIL ];
```

ScrollData is passed to **Create** and **SetScrollData** to specify the desired scrolling. **displayHorizontal** indicates whether the shell should have a horizontal scrollbar. **displayVertical** indicates whether the shell should have a vertical scrollbar. **arrowScroll** is called when the user points at the arrows or at the plus or minus signs. **thumbScroll** is called when the user thumbs. These procedures are expected to perform the appropriate scroll, probably by moving body windows with **Window.Slide**. If either **arrowScroll** or **thumbScroll** are NIL, **StarWindowShell** provides default scrolling procedures (**VanillaArrowScroll** and **VanillaThumbScroll**) that operate as described above. **moreScroll** is called when **VanillaArrowScroll** or **VanillaThumbScroll** needs more body windows to be supplied by the client. The client should never need to supply both an **arrowScroll** and a **moreScroll**.

```
ArrowScrollProc: TYPE = PROCEDURE [
   sws: Handle,
   vertical: BOOLEAN,
   flavor: ArrowFlavor,
   arrowScrollAction: ArrowScrollAction ← go];

ArrowFlavor: TYPE = (pageFwd, pageBwd, forward, backward};
```

An **ArrowScrollProc** is called whenever the user points at an arrow or the plus or minus sign in a scrollbar. The **ArrowScrollProc** is expected to scroll the contents of **sws** as appropriate. **vertical** indicates whether to scroll vertically (TRUE) or horizontally (FALSE). **flavor** indicates what type of scrolling the user requested. **pageFwd** means the user pointed at the plus sign (**vertical** = TRUE) or the right margin symbol (**vertical** = FALSE). **pageBwd** means the user pointed at the minus sign (**vertical** = TRUE) or the left margin symbol (**vertical** = FALSE). **forward** means the user pointed at the up-pointing arrow (**vertical** = TRUE) or the left-pointing arrow (**vertical** = FALSE). **backward** means the user pointed at the down-pointing arrow (**vertical** = TRUE) or the right-pointing arrow (**vertical** = FALSE). The **ArrowScrollProc** will be called repeatedly as long as the user has the mouse button down over one of the arrows, thus producing continuous scrolling. Note:

EnumerateBodiesInIncreasingY and **EnumerateBodiesInDecreasingY** are quite useful when scrolling body windows (see the section on body windows).

ArrowScrollAction: TYPE = {start, go, stop};

start indicates the user just buttoned down. **go** indicates the user still has the button down. **stop** indicates the user just buttoned up. This allows clients to scroll body windows without repainting until the **ArrowScrollProc** is called with **arrowScrollAction = stop**.

```
ThumbScrollProc: TYPE = PROCEDURE [
  sws: Handle, vertical: BOOLEAN, flavor: ThumbFlavor, m, outOfN: INTEGER];
```

ThumbFlavor: TYPE = {downClick, track, upClick};

A **ThumbScrollProc** is called whenever the user points in the thumbing part of the vertical scrollbar. **vertical** is always TRUE (horizontal thumbing is not currently allowed). **flavor** indicates whether the user has just buttoned down (**downClick**), is moving the mouse with the button down (**track**), or has just released the button (**upClick**). Usually, the actual scrolling does not take place until the **upClick**. **downClick** and **track** give the client an opportunity to display information to the user, such as what page number the current cursor location corresponds to (see **Cursor.NumberIntoCursor**). **m** and **outOfN** indicate where the cursor is with respect to the entire extent of the thumbing area. For example, if **m** = 200 and **outOfN** = 400, the user wants to thumb to the middle of the entire object. Note: **EnumerateBodiesInIncreasingY** and **EnumerateBodiesInDecreasingY** are quite useful when scrolling body windows (see the section on body windows).

PaintThumbFeedBack: PROC [sws: Handle, offset: Percent, portion: Percent ← 0];

EraseThumbFeedBack: PROC [sws: Handle];

GetScrollbar: PROC [sws:Handle, vertical: BOOLEAN ← TRUE] RETURNS [Window.Handle];

Percent: TYPE = INTEGER [0..100];

PaintThumbFeedback, etc. support the painting of feedback into the thumbing region of a scrollbar. These items are defined in StarWindowShellExtra3.mesa.

PercentOf: PROC [v: INTEGER, p: Percent] RETURNS [INTEGER];

PercentOf expresses **p** in terms of v. For example, **m ← PercentOf[OutOfN, offset]** .

Percentage: PROC [part, full: INTEGER] RETURNS [Percent];

Percentage returns the percentage of **part** to **full**. For example, **offset ← Percentage[m, OutOfN]**.

PaintThumbFeedback, etc. support the painting of feedback into the thumbing region of a scrollbar. These items are defined in StarWindowShellExtra3.mesa.

```
MoreScrollProc: TYPE = PROCEDURE [
  sws: Handle, vertical: BOOLEAN, flavor: MoreFlavor, amount: CARDINAL];
```

```
MoreFlavor: TYPE = {before, after};
```

A **MoreScrollProc** is called by the default **StarWindowShell** scrolling procedures, **VanillaArrowScroll** and **VanillaThumbScroll**, whenever more body windows are needed to continue scrolling. This is used when the client has several body windows butted against one another. When the user points at an arrow, **VanillaArrowScroll** moves all the body windows a small amount. When the user point at plus or minus, **VanillaArrowScroll** moves all the body windows by one interior window's height. When the user thumbs, **VanillaThumbScroll** moves all the body windows to an appropriate place based on the combined overall height of the body windows. However, the client often does not have the entire extent of the object displayed in these body windows but rather wants to tack on new body windows on each end as these body windows are scrolled off. This is when the client's **MoreScrollProc** is called. **vertical** indicates whether the user was scrolling vertically or horizontally. **flavor** indicates whether to tack on more body windows **before** (that is, the user was scrolling down for **vertical** = TRUE, right for **vertical** = FALSE), or **after** (that is, the user was scrolling up for **vertical** = TRUE, left for **vertical** = FALSE). **amount** indicates how much extra body window is needed, in screen dots.

The client's **moreScroll** procedure is responsible for adding and deleting body windows from the shell. The case being handled is that in which the client has a large number of pages to display to the user and wishes to manifest only a few. Then we need to handle the case in which system scrolling would make a non-manifest page visible, and there is no body window for it. Whenever the system is about to perform a scroll function, it checks to see if the scroll action would move the visible portion of the bodies off the end of the existent body windows. If so, it calls a non-nil client **MoreScrollProc**, indicating how much more body window may be displayed. The client may augment the collection of body windows or not. The system routines will not scroll past the end of the body windows. The client may also use this opportunity to garbage-collect body windows that have been scrolled off the other end and are no longer visible.

```
noScrollData: ScrollData ← [];
```

noScrollData indicates no scrollbars at all.

```
vanillaScrollData: ScrollData ← [
  displayHorizontal: FALSE,
  displayVertical: TRUE,
  arrowScroll: NIL, -- actually VanillaArrowScroll
  thumbScroll: NIL,-- actually VanillaThumbScroll
  moreScroll: NIL ];
```

vanillaScrollData is the default for the **scrollData** parameter to **Create**. It indicates avertical scrollbar with the **StarWindowShell.VanillaXXXScroll** procedures described above.

```
GetScrollData: PROCEDURE [sws: Handle] RETURNS [scrollData: ScrollData];
```

GetScrollData returns the current **ScrollData** associated with **sws**. May raise **Error** [**notASWS**].

```
SetScrollData: PROCEDURE [sws: Handle, new: ScrollData]
  RETURNS [old: ScrollData];
```

SetScrollData sets the current **ScrollData** for **sws** and returns the previous. May raise **Error [notASWS]**.

VanillaArrowScroll: ArrowScrollProc;

VanillaThumbScroll: ThumbScrollProc;

The default scrolling procedures that **StarWindowShell** provides are exported here. This allows a client to insert its own scroll procedures, check for certain conditions that it wants to handle, and call **StarWindowShell** to do the scrolling for other conditions.

50.2.6 Push, Pop, etc.

The **StarWindowShell** returned by **Create** is not displayed on the screen; that is, is not inserted into the visible window tree. A **StarWindowShell** may be inserted into the window tree by calling **Push**. This is usually not done by the client but rather by some other part of ViewPoint, such as the desktop implementation. For example, when the user selects an icon and presses OPEN or PROPS, the application (actually the application's **Containee.GenericProc**) creates a **StarWindowShell** and returns it. The desktop implementation then displays the **StarWindowShell** by doing a **Push**.

A **StarWindowShell** is removed from the screen by calling **Pop**. Clients almost never call **Pop**. Rather, **StarWindowShell** calls **Pop** when the user selects Close, or PropertySheet calls **Pop** when the user selects Done or Cancel.

Push allows one shell to be pushed on top of another shell, thus providing the illusion of "open-within." For example, Star folders and file drawers use this illusion. **StarWindowShell** has provisions for a shell to display commands in the header of the shells pushed on top of it. (See the section one **Pushee** commands.) Most clients will not make use of this feature of **Push**, because the **ContainerWindow** interface takes care of this for applications that appear as a list of items that may be opened. Fine point: We simplify things here by replacing the entire shell. When the shell on top is closed, the shell below still exists and is simply redisplayed.

```
Push: PROCEDURE [
  newShell: Handle, topOfStack: Handle ← NIL,
  poppedProc: PoppedProc ← NIL ];
```

Push displays **newShell** by inserting it into the visible window tree. If **topOfStack** is NIL, **newShell** is placed directly on the desktop. If **topOfStack** is *not* NIL, then **newShell** is pushed on top of **topOfStack** and **topOfStack** is removed from the display (but see the fine point below). If **topOfStack** is not NIL, it must be currently visible (that is, does not have another shell **Push**ed on top of it). If **poppedProc** is not NIL, it is called when **newShell** is **Popped**. The **poppedProc** must either sleep the shell or destroy the shell, usually by calling **SleepOrDestroy**. If **poppedProc** is NIL, **newShell** will be destroyed when it is **Popped**. Note that **Push** can be called repeatedly with **topOfStack** being the **newShell** from the previous call, thus producing a stack of **StarWindowShells**. May raise **Error [notASWS]**.

Fine point: For open-within, we are experimenting with opening the **newShell** overlapping the **topOfStack** shell, allowing the user to look at the container and the thing contained at the same time. This has some rather complex implications with respect to having two views of the same things, being able to open several contained items at once, and so forth.

PushedMe: PROCEDURE [pushee: Handle] RETURNS [pusher: Handle];

PushedMe returns the next lower shell below **pushee** in the stack (NIL if none). **PushedMe** is defined in StarWindowShellExtra.mesa.

PushedOnMe: PROCEDURE [pusher: Handle] RETURNS [pushee: Handle];

This procedure returns the next higher shell above **pusher** in the stack (NIL if none). **PushedOnMe** is defined in StarWindowShellExtra.mesa.

```
PoppedProc: TYPE = PROCEDURE [popped, newShell: Handle,
   popOrSwap: PopOrSwap ← pop];
```

```
PopOrSwap: TYPE = {pop, swap};
```

popped is the shell that is being taken out of the visible window tree. **newShell** is the shell that becomes visible because of **popped** being popped. It is NIL if **popped** was not opened within another window. **popOrSwap** indicates the action that caused the shell to be popped, either StarWindowShell.Pop or StarWindowShell.Swap.

Pop: PROCEDURE [popee: Handle] RETURNS [Handle];

Pop removes **popee** from a stack of shells and returns the shell that is now on top of the stack. If **popee** was **Pushed** with a **poppedProc**, this **poppedProc** is called. If **popee** is not the top of a stack, then all shells above it in the stack are **Popped**. May raise **Error [notASWS]**.

```
Swap: PROCEDURE [
   new, old: Handle,
   poppedProc: PoppedProc ← NIL] ;
```

Swap replaces **old**, which must be the top of a stack, with **new**. Equivalent to a **Pop** followed by a **Push**, but with a lot less screen flashing. May raise **Error [notASWS]**. Folder uses this procedure when doing a "Show Previous".

Replace: PROCEDURE [new, old: Handle];

Replaces the **old** shell with **new** without calling **old's PoppedProc**. **old's PoppedProc** becomes the **PoppedProc** for **new**. Fine point: This procedure is currently exported through StarWindowShellExtra.. TTY uses this procedure when going from the option sheet to the tty window. (The option sheet's MenuItemsProc calls. StarWindowShellExtra.Replace then returns ok = FALSE to MenuItemsProc's caller.)

Note that the application is responsible for destroying the extra shell, since the PoppedProc (when it does eventually get called) will destroy only the shell actually being popped. Thus, in general, the application should call SWS.Destroy on the psheet shell sometime after doing the Replace.

50.2.7 Limit and Adjust Procs

Limit and Adjust procs are client-supplied procedures that allow a client to get control whenever a **StarWindowShell** is going to change size or location. A **LimitProc** gives the client control over the size and placement of a shell. An **AdjustProc** gives the client an opportunity to fix up the data structures and display for the shell's body window(s).

LimitProc: TYPE = PROCEDURE [sws: Handle, box: Window.Box] RETURNS [Window.Box];

GetLimitProc: PROCEDURE [sws: Handle] RETURNS [LimitProc];

SetLimitProc: PROCEDURE [sws: Handle, proc: LimitProc] RETURNS [old: LimitProc];

Whenever the size or location of a shell is going to change, the client's **LimitProc** is called. This allows the client to exercise veto or modification rights over the size and location of a **StarWindowShell**. This is useful, for example, to prohibit a shell from becoming smaller than some certain size or from being moved completely off the screen. **box** is the requested size of the shell. The **LimitProc** should return the desired size of the shell. The **LimitProc** is called before the **AdjustProc**. The interior box of the shell box returned by the **LimitProc** is passed to the **AdjustProc**. **GetLimitProc** and **SetLimitProc** may raise **Error [notASWS]**.

StandardLimitProc: LimitProc;

A **StandardLimitProc** is provided that keeps shells on the screen and keeps them from getting too small.

AdjustProc: TYPE = PROCEDURE [sws: Handle, box: Window.Box, when: When];

When: TYPE = {before, after};

GetAdjustProc: PROCEDURE [sws: Handle] RETURNS [AdjustProc];

SetAdjustProc: PROCEDURE [sws: Handle, proc: AdjustProc] RETURNS [old: AdjustProc];

The **AdjustProc** is called whenever the shell is going to change size. It is called both before and after the window's size is changed. The **box** passed to the **AdjustProc** is the *interior* window's box (the client's viewing region in the shell). The **when** parameter indicates whether the current call is before or after the window's size has been changed. An **AdjustProc** is for those clients whose body window display depends on the size of the surrounding shell. For example, if the body window sticks out of the interior of the shell and the user must scroll the body window horizontally to see all the contents, then no **AdjustProc** is needed. If, on the other hand, the content of the body window is always kept visible regardless of the size of the shell (by wrapping the contents around as in the Tajo **FileWindow** editor), then the client needs an **AdjustProc**. **GetAdjustProc** and **SetAdjustProc** may raise **Error [notASWS]**.

50.2.8 Displayed StarWindowShells

EnumerateDisplayed: PROCEDURE [proc: ShellEnumProc] RETURNS [Handle ← [NIL]];

**EnumerateDisplayedOfType: PROCEDURE [ShellType, proc: ShellEnumProc]
RETURNS [Handle ← [NIL]];**

**EnumerateMyDisplayedParasites: [sws: Handle, proc: ShellEnumProc]
RETURNS [Handle ← [NIL]];**

```
ShellEnumProc: TYPE = PROCEDURE [victim: Window.Handle]
   RETURNS [stop: BOOLEAN ← FALSE];
```

These procedures enumerate visible **StarWindowShells**. Each one returns the last shell incurred in the enumeration if the **ShellEnumProc** returns TRUE; otherwise, they return NIL. **EnumerateMyDisplayedParasites** may raise **Error [notASWS]**.

50.2.9 Errors

```
Error: ERROR[code: ErrorCode];

ErrorCode: TYPE = {desktopNotUp, notASWS, notStarStyle, tooManyWindows};
```

50.3 Usage/Examples

50.3.1 Example 1

```
-- Create a StarWindowShell - simple case

CreateShell: PROCEDURE RETURNS [StarWindowShell.Handle] =
BEGIN

   another: XString.ReaderBody ← XString.FromSTRING["Another"L];
   repaint: XString.ReaderBody ← XString.FromSTRING["Repaint"L];
   post: XString.ReaderBody ← XString.FromSTRING["Post A Message"L];
   sampleTool: XString.ReaderBody ← XString.FromSTRING["Sample Tool"L];

   -- Create the StarWindowShell
   -- Note: Since a zone was not supplied, one will be created (and destroyed) for you.
   shell: StarWindowShell.Handle = StarWindowShell.Create [name: @sampleTool];

   -- Create a body window inside the StarWindowShell
   body: Window.Handle = StarWindowShell.CreateBody [
      sws: shell,
      box: [ [0,0], bodyWindowDims ],
      repaintProc: Redisplay,
      bodyNotifyProc: NotifyProc ];

   -- Create some menu items
   z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
   items: ARRAY [0..3) OF MenuData.ItemHandle ← [
      MenuData.CreateItem [zone: z, name: @another, proc: MenuProc],
      MenuData.CreateItem [zone: z, name: @repaint, proc: RepaintMenuProc],
      MenuData.CreateItem [zone: z, name: @post, proc: Post] ];
   myMenu: MenuData.MenuHandle = MenuData.CreateMenu [
      zone: z,
      title: NIL,
      array: DESCRIPTOR [items]];
   StarWindowShell.SetRegularCommands [sws: shell, commands: myMenu];
```

```
        -- The calling procedure will call StarWindowShell.Push on the returned shell to make
        -- the shell visible.
        RETURN [shell];

        END; -- of CreateShell
```

50.3.2 Example 2

```
    -- Create a StarWindowShell using a client supplied Zone
    -- There are various Heap checking tools for determining the
    -- appropriate parameters to be used when creating the Zone.
    -- Performance can be dramatically affected by these parameters.

    CreateShell:PROCEDURE RETURNS [StarWindowShell.Handle] =
    BEGIN

        myZone: UNCOUNTED ZONE = Heap.Create[initial: 5];
        sampleTool: XString.ReaderBody ← XString.FromSTRING["Sample Tool"L];

        -- Create the StarWindowShell
        -- Now a zone and a TransitionProc were specified.
        shell: StarWindowShell.Handle = StarWindowShell.Create [
         name: @sampleTool,zone: myZone, transitionProc: MyTransitionProc];

        -- Need to define the DestroyZoneProc so we can destroy the zone at the end.
        --Since we are using Heap to create the zone, we will use
        -- the DefaultDestroyZoneProc which does not examine client data, so leave it NIL.
        [] ← StarWindowShellExtra5.SetDestroyZoneProc[
         sws: shell, proc: DefaultDestroyZoneProc];

        -- The rest of this procedure is exactly as shown in EXAMPLE 1

        END; -- of CreateShell

    MyTransitionProc: StarWindowShell.TransitionProc =
    BEGIN

        --This procedure will do various tasks when the shell is Created or Destroyed.
        -- It could allocate data structures (state = awake) or
        -- It could deallocate data used only while the shell was open (state = sleeping).
        -- It could deallocate all data created for this instance of the tool (state = dead).
        -- This example handles only the destruction (state = dead) state.

        SELECT state FROM

        awake = > NULL;

        sleeping = > NULL;

        dead = > {
```

```
        -- The zone and shell is still around. Client allowed to muck with
        -- the data contained therein. For example, copy it into a file
        z: UNCOUNTED ZONE ← StarWindowShell.GetZone[sws];

        -- Do some miscellaneous cleanup.
        -- NOTE: Cannot delete the Zone - StarWindowShell.Destroy is not finished yet!
        };

    ENDCASE;

    END; -- of MyTransitionProc
```

50.3.3 Example 3

```
-- Destroy the client supplied Zone in the background
-- This example does as much as possible in FORKed procedures
-- so that the shell can be closed as fast as possible.
-- Note that Process.TooManyProcesses is always handled when FORKing.

CreateShell:PROCEDURE RETURNS [StarWindowShell.Handle] =
BEGIN

    myZone: UNCOUNTED ZONE = Heap.Create[initial: 5];
    sampleTool: XString.ReaderBody ← XString.FromSTRING["Sample Tool"L];

    -- Create the StarWindowShell
    -- A zone and a TransitionProc were specified.
    shell: StarWindowShell.Handle = StarWindowShell.Create [
     name: @sampleTool,zone: myZone, transitionProc: MyTransitionProc];

    -- Do NOT register the DestroyZoneProc yet!

    -- The rest of this procedure is exactly as shown in EXAMPLE 1

    END; -- of CreateShell

DestroyZoneProcData: TYPE = RECORD [process: PROCESS];

MyTransitionProc: StarWindowShell.TransitionProc =
BEGIN

    --This procedure will do cleanup in the background.

    SELECT state FROM

    awake = > NULL;

    sleeping = > NULL;

    dead = > {

        -- The zone and shell is still around.
        -- Do some miscellaneous cleanup in the background
        clientData: LONG POINTER TO DestroyZoneProcData;
        z: UNCOUNTED ZONE ← StarWindowShell.GetZone[sws];

        -- Do the cleanup in another process. If this is impossible,
        -- then recover as gracefully as possible.
        -- Do not change Process.Priority of the current (NOTIFIER)
        -- process - dire results possible.
        p: PROCESS ← FORK Cleanup[z, TRUE ! Process.TooManyProcesses => {
                Cleanup[z, FALSE];
                Heap.Delete[z];
                EXIT}];

        -- Need to remember the process that is doing Cleanup.
        clientData ← z.NEW[DestroyZoneProcData ← [p]];

        -- Call SetDestroyZoneProc now that we have the clientData for it.
        [] ← StarWindowShellExtra5.SetDestroyZoneProc[
                sws, MyDestroyZoneProc, clientData];

        };

    ENDCASE;

    END; -- of MyTransitionProc

Cleanup: PROC [z: UNCOUNTED ZONE, backgroundOK: BOOLEAN] =
BEGIN

    -- Do whatever cleanups are necessary.

    -- Cannot delete the zone.
    -- StarWindowShell is not yet finished with it.

    -- If not in the NOTIFIER then ok to change priority.
    IF backgroundOK THEN Process.SetPriority[Process.priorityBackground];

    END; -- of Cleanup

MyDestroyZoneProc: StarWindowShellExtra5.DestroyZoneProc =
BEGIN

    -- Want to return as quickly as possible so FORK the actual destroy.
    -- Guaranteed that StarWindowShell is finished with the zone at
    -- this point. Now it is safe to destroy the zone.
    Process.Detach[
    FORK DoDestroyInBackground[z, clientData, TRUE
        ! Process.TooManyProcesses => {
                DoDestroyInBackground[z, clientData, FALSE];
                CONTINUE}]];

    END; -- of MyDestroyZoneProc
```

```
DoDestroyInBackground: PROC [z: UNCOUNTED ZONE, clientData: LONG POINTER TO
DestroyZoneProcData, backgroundOK: BOOLEAN] =
BEGIN

    -- Grab my data out of heap before deleting it.
    -- This is where I stored the Cleanup process.
    destroyZoneProcData: DestroyZoneProcData = clientData↑;

    -- If not in the NOTIFIER then ok to change priority.
    IF backgroundOK THEN Process.SetPriority[Process.priorityBackground];

    -- Wait until Cleanup process is done with zone
    JOIN destroyZoneProcData.process;

    -- Now delete the zone.
    Heap.Delete[z];

    END; -- of DoDestroyInBackground
```

50.4 Index of Interface Items

51

Subwindow Overview

51.1 Overview

The subwindow package provides a facility that clients can use to incorporate subwindows into their applications. We provide the capability to make subwindows scrollable horizontally, vertically or both, adjustable horizontally, vertically or both, moveable, and top/bottomable. The initial version includes the implementation of three specific types of subwindows: form subwindows, message subwindows and body subwindows. This allows a client to incorporate a form subwindow (etc.) into an application. A facility to allow client defined subwindow types is also provided. Two levels of use of subwindows are supported. Clients can subdivide a shell and expect the subwindows to adjust in harmony with each other or alternately, place a subwindow in an isolated environment unrelated to any other subwindow.

The interfaces that make up the subwindow package will be described here in terms of typical usage starting with the most common use and working down to the least common. Details about the various interfaces can be found in the individual chapters. The interfaces are: **Subwindower, StarWindowShellExtra5, SubwindowManager, SubwindowFriends, AdjustableWindow, Scrollbar, BodyWindowParent.**

51.2 Summary of Interfaces

This is a brief description from the highest level interfaces to the lowest. We propose that Subwindower and StarWindowShellExtra5 are the only truly public interfaces. The rest are friends.

Subwindower
Highest level interface, real easy to use. Given a window, it allows the predefined types of subwindows to be created inside it, such as FormWindows, MessageWindows, BodyWindows. Subwindower is a tiny impl that uses SubwindowManager and existing interfaces like FormWindow and MessageWindow.

StarWindowShellExtra5
Given a shell, gives you back the window that can be passed to Subwindower as the swmanager. It 'just-fits' inside the shell viewing region and adjusts subwindows properly when the shell is adjusted, etc.

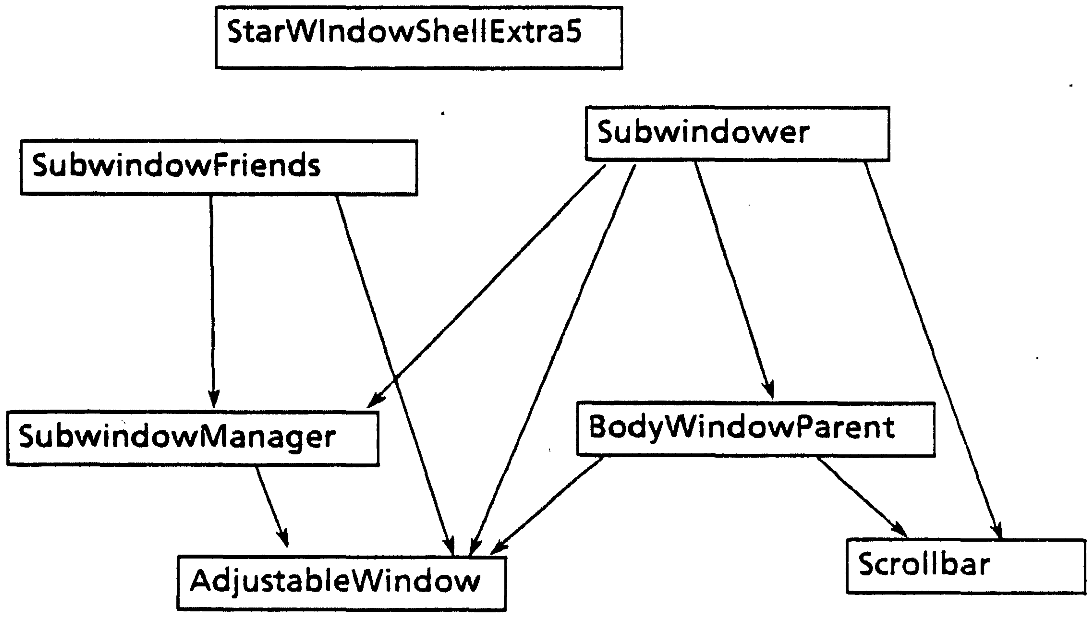
SubwindowManager
Provides the basic subwindow manager functions, e.g. Creating subwindows, adding and removing subwindows, enumerating them, adjustment UI, etc. Subwindower uses it to create the predefined types.

SubwindowFriends
Provides for client defined subwindow types. For example, FormWindow calls in to register its adjust procs etc to manage form subwindows. Clients can register their own types.

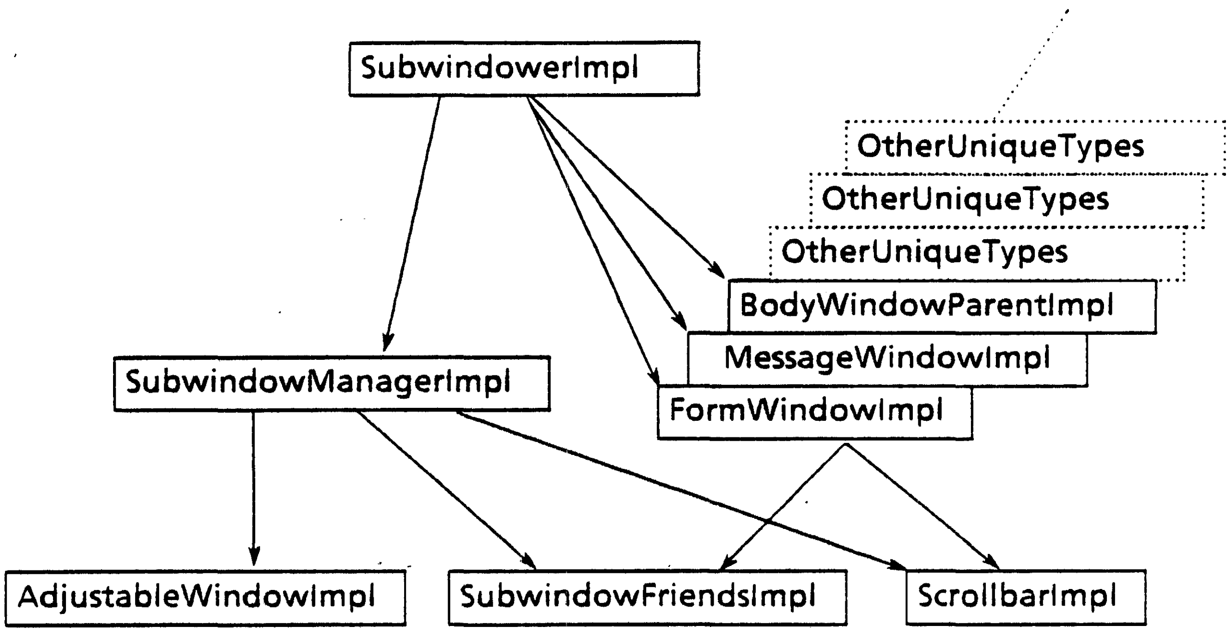
BodyWindowParent
Predefined subwindow type. Exports CreateBody, GarbageCollectBodiesProc, etc. analagous to StarWindowShell, but the whole assembly (body windows, parent clipping window, scrollbars, etc.) is a subwindow. FormWindows as subwindows use this.

AdjustableWindow
Makes an arbitrary window become shrinkable, growable, moveable by the user (client has all sorts of options). Used by SubwindowManager. This is where AdjustProc and LimitProc are defined.

Scrollbar
For attaching scrollbars to a window. Client supplies several scroll procs. Used by Predefined types.



Subwindow Interface Compile time Dependencies



Subwindow Impl Dependencies

51.3 Subdividing a Shell

A client can easily divide a shell into subwindows that will adjust in harmony with each other. A subwindow may only grow as big as its neighbors minimum height will allow or as small as its own minimum height restrictions. It is expected that most clients of subwindows will require only the procedures described in §51.3.1.

51.3.1 Creating Predefined SWs

The great majority of clients will be interested in dividing a shell into several subwindows that resize in a coordinated manner. The following code segment divides a shell into a message subwindow, form subwindow and a body subwindow. The message window in this example does not have a horizontal scrollbar.

```
CreateMyToolWindow: PROCEDURE = {
  myTool: XString.ReaderBody ← XString.FromSTRING["My Tool"L];
--create a shell without scrollbars
  shell: StarWindowShell.Handle ← StarWindowShell.Create [
    name: @myTool, scrollData: [FALSE,FALSE,NIL,NIL,NIL] ];
  z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
--get the shell's subwindow manager
  swManager: Window.Handle ← StarWindowShellExtra5.ManagerFromShell [
    sws: shell];
--create an adjustable message subwindow with a vertical scrollbar only
  msgSW: Window.Handle ← Subwindower.MakeMessageSW[
    swmanager: swManager, size: some#, horizScrollbar: FALSE, zone: z];
--create a form subwindow that is adjustable and has both scrollbars
  formSW: Window.Handle ← Subwindower.MakeFormSW[
    swmanager:swManager, size: some#, makeItemsProc: MakeItems, zone: z];
--create a body subwindow that is adjustable and has both scrollbars
  bodySW: Window.Handle ← Subwindower.MakeBodySW[
    swmanager:swManager,size: some#, zone: z];
  body: Window.Handle ← BodyWindowParent.CreateBody[bodySW,
    displayProc: DisplayProc, notify: NotifyProc];
  StarWindowShell.Push [shell];
  };
```

A subwindow manager that just fills the available interior space in the shell is created automatically when you create a shell. This manager will continue to "just fit" in the shells interior during subsequent resizes of the shell. Most clients are expected to use this manager.

Subwindower provides the highest level facility for creating subwindows in a window. Given a window, it allows predefined types of subwindows to be created inside it. It is typically used in conjunction with StarWindowShellExtras.ManagerFromShell. Three types of subwindows are currently provided by Subwindower. They are MessageWindows, FormWindows, and BodyWindows. More types are expected to be added later.

A client of Subwindower.MakeBodySW will also be interested in BodyWindowParent.CreateBody for creating body windows to go inside the bodyParent that is returned by MakeBodySW. Any number of body windows may be linked inside of the bodyParent.

51.3.2 Client Defined SWs

Clients interested in creating their own unique subwindow type will use the SubwindowFriends interface. The predefined types in Subwindower are implemented this way.

Clients can create their own subwindow types by using SubwindowFriends.UniqueType, SubwindowFriends.SetSWProcs and SubwindowManager.MakeSW. **UniqueType** returns a unique identifier that is used later by other procedures. **SetSWProcs** takes a unique type and several procedures that are to be called whenever **MakeSW** is called with that unique type. **MakeSW** can be called with any uniqueType or with vanilla to create a subwindow of that type. A vanilla subwindow uses the standard procs provided in SubwindowFriends which creates a viewer with scrollbars that do simple scrolling on descendants of the viewer. (No descendants are provided. This is up to the client.)

See the SubwindowManager and SubwindowFriends chapters §51.3 for Sample code.

51.3.3 Inserting and Deleting SWs

It is possible for a client to add and subtract existing subwindows from their parent by using several SubwindowManager procedures. If a client wishes to create a new subwindow and insert it above some other subwindows within the manager he can call **MakeXXXSW** (where XXX is Form, Message or Body) or **MakeSW** with swmanager = FALSE (creating an orphaned subwindow), then at some later time call SubwindowManager .AppendSW, InsertSW, RemoveSW, or SwapSW to place it in the desired location in the swmanager.

See the SubwindowFriends chapter §53.3 for Sample code.

51.4 Independent SWs

Another use of subwindows is available that allows a client to create a subwindow from any given window. Example: turn a FormWindow windowItem into a subwindow. See figure 3. In this case the subwindow resize would be controlled by FormWindow. If the user is allowed to adjust the shape of the subwindow, FormWindow will adjust any other items in the form window to accomodate the new size of the window item/subwindow. Uses which do not include FormWindow or swManager will have to provide any coordination of resizes themselves.

An independent subwindow may specify if it is to be moveable or top/bottomable as well as growable. Move, top and bottom all refer to relative location of sibling windows. In other words, it is possible for a client to show the user several overlapping (independent) subwindows. The user could move, top or bottom these subwindows with respect to one another in the same fashion as moving and top/bottoming shells.

A client can create an independent subwindow by calling AdjustableWindow.Create passing in the window he wishes to turn into a subwindow. Scrollbars can be added to this subwindow by calling Scrollbar.Attach passing scrollProcs as desired.

51.5 Usage/Examples

Figures 1-4 on the following pages show a number of typical subwindow configurations. The relevant lines of code are outlined below.

Figure 1 (Figures 1,2 and 4 are very similar, therefore separate code is not shown for each)

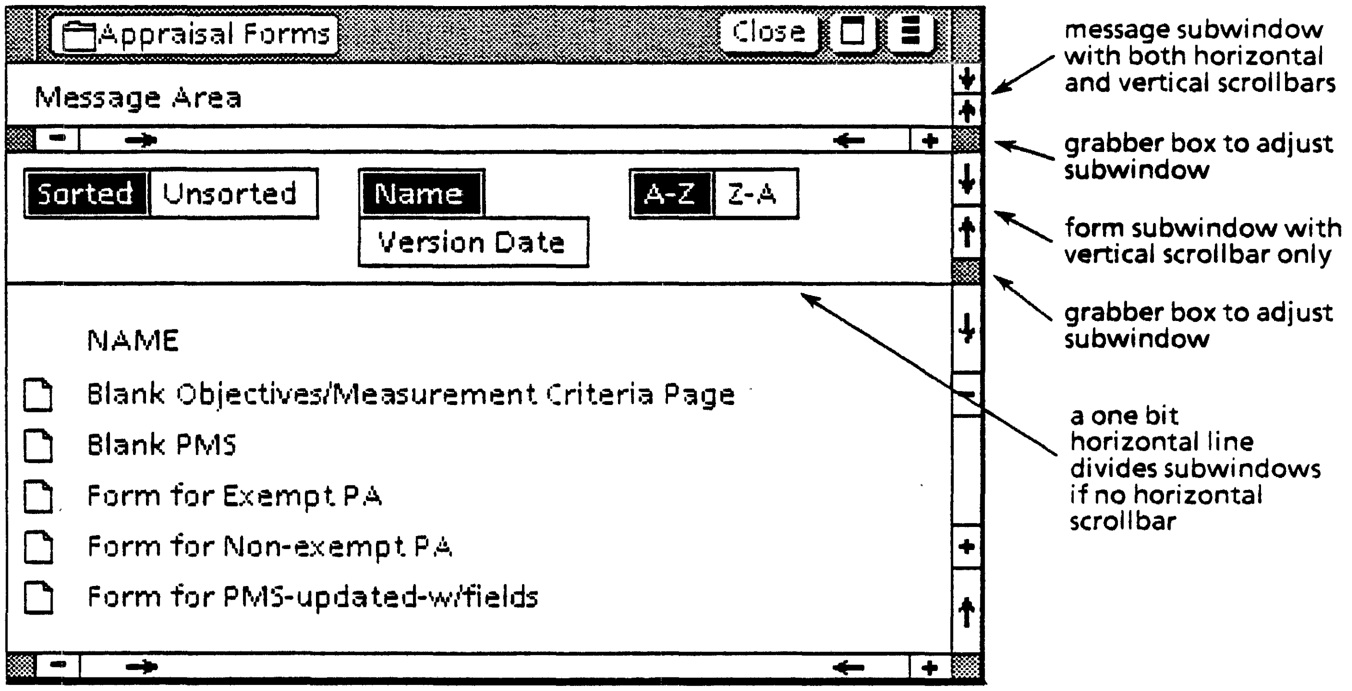
```
   shell: StarWindowShell.Handle ← StarWindowShell.Create [
      name: name, scrollData: [FALSE,FALSE,NIL,NIL,NIL] ];
   swManager: Window.Handle ← StarWindowShellExtra5.ManagerFromShell[ shell];
   msgSW: Window.Handle ← Subwindower.MakeMessageSW[
      swmanager: swManager, size: small, zone: z];
   formSW: Window.Handle ← Subwindower.MakeFormSW[
      swmanager:swManager, size: medium, horizScrollbar: FALSE,
      makeItemsProc: MakeItems, zone: z];
--the next sw might be a "vanilla" subwindow created as follows:
   mySW: Window.Handle ← SubwindowManager.MakeSW[
      swmanager:swManager,size: large, zone: z];
--OR To implement and use your own unique type of subwindow :
   myType ← SubwindowFriends.UniqueSWType[]
    SubwindowFriends.SetSWProcs[myType,MyAttachScrollbarsProc];
   mySW ← SubwindowManager.MakeSW[type: myType
      swmanager:swManager, size: large, zone: z];
```

Figure 3

```
   --create a shell with a form window inside it
   --in the MakeItemsProc create a windowItem to use for the log subwindow
   --then turn that window item into a subwindow

   --make the window item an adjustable window
   AdjustableWindow.Create[windowItem,WDimsChangeProc, zone,
      MyLimitProc, [FALSE,TRUE,FALSE,TRUE], FALSE, FALSE];
   --implement some variety of AttachScrollbarsProc or call the Standard version
   SubwindowFriends.StandardAttachScrollbars[...];

WDimsChangeProc:AdjustableWindow.AdjustProc. = {
   SELECT when FROM after = >
      FormWindow.SetWindowItemSize[form, wItemkey, box] }
   ENDCASE }
```



Appraisal Forms

Close

Message Area

Sorted | Unsorted

Name | Version Date

A-Z | Z-A

NAME

Blank Objectives/Measurement Criteria Page

Blank PMS

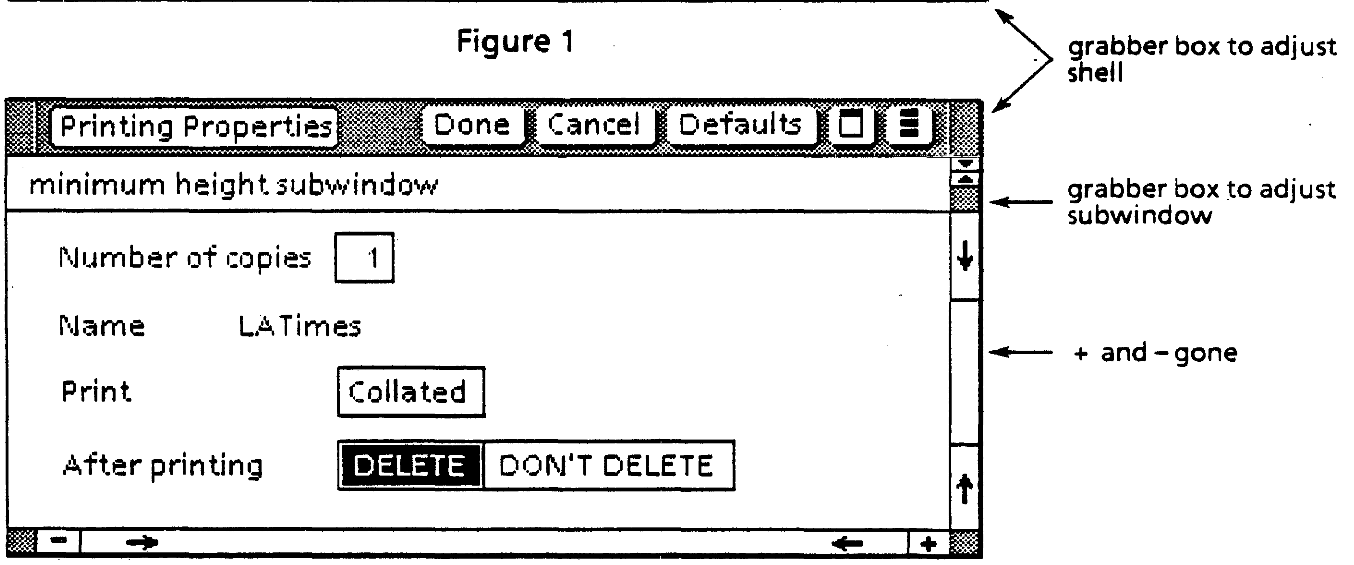
Form for Exempt PA

Form for Non-exempt PA

Form for PMS-updated-w/fields

message subwindow with both horizontal and vertical scrollbars
grabber box to adjust subwindow
form subwindow with vertical scrollbar only
grabber box to adjust subwindow
a one bit horizontal line divides subwindows if no horizontal scrollbar

Figure 1



Printing Properties

Done | Cancel | Defaults

minimum height subwindow

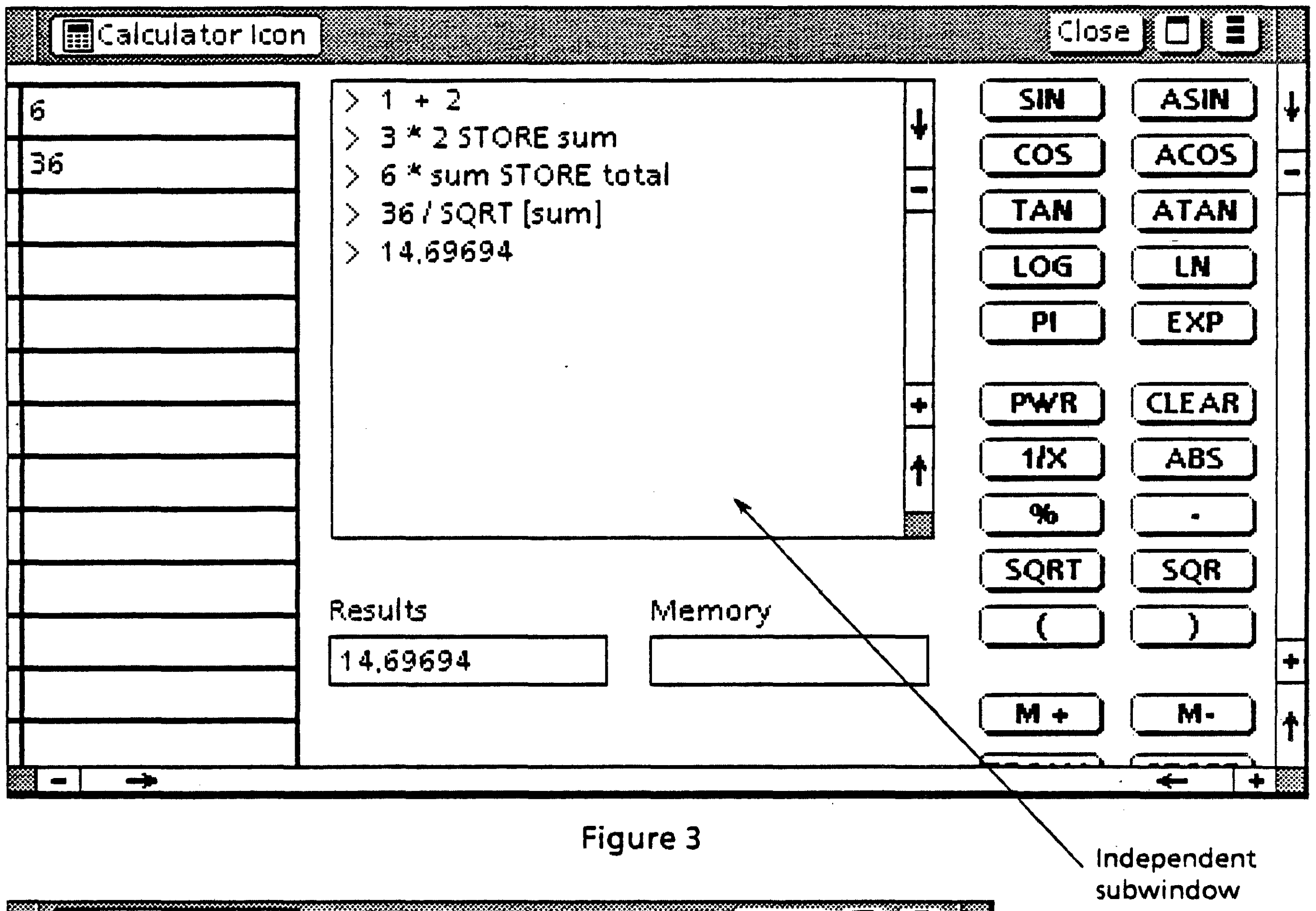
Number of copies 1

Name LATimes

Print Collated

After printing DELETE | DON'T DELETE

Figure 2



```
6
36
```

```
> 1 + 2
> 3 * 2 STORE sum
> 6 * sum STORE total
> 36 / SQRT [sum]
> 14.69694
```

Results: 14.69694

Figure 3

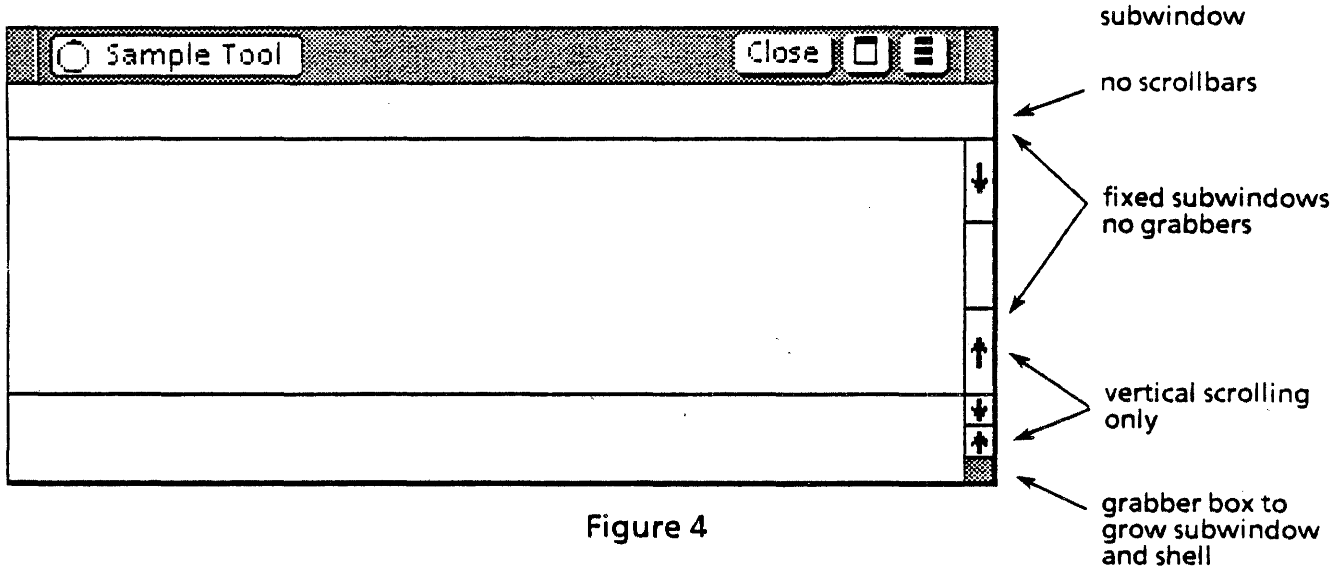

Sample Tool
Close
no scrollbars
fixed subwindows no grabbers
vertical scrolling only
grabber box to grow subwindow and shell

Figure 4

52

Subwindower

52.1 Overview

Subwindower provides the highest level facility for creating subwindows in a window. Given a window, it allows predefined types of subwindows to be created inside it, e.g. BodyWindows, FormWindows, MessageWindows. It is typically used in conjunction with **StarWindowShell.ManagerFromShell**. Fine point: in BWS 4.3, **ManagerFromShellis** in **StarWindowShellExtra5**. See the chapters on **FormWindow**, **MessageWindow** and **BodyWindowParent** for the specifics on each type of subwindow. For a comprehensive view of all the subwindow interfaces and their intended use, see the Subwindow Overview chapter.

52.2 Interface Items

52.2.1 Making subwindows

Three types of subwindows are currently provided by **Subwindower**. They are MessageWindows, FormWindows, and BodyWindows. Fine Point: more types are expected to be added later. Each is identical to its non-subwindow counterpart in all functional respects. These window types can be created using **MakeFormSW**, **MakeMessageSW** and **MakeBodySW** described in the following sections.

A number of parameters to each **MakeXXXSW** procedure are identical and are described here, rather than with each procedure.

swmanager refers to the parent window that will contain the subwindow(s). (Note: a swmanager that manages the adjustment of adjacent subwindows may be obtained by calling **StarWindowShellExtra5.ManagerFromShell**.)

size specifies the size of the subwindow (see Figure 1). If scrollbars are attached, the viewing region within the subwindow is reduced by the width of a vertical scrollbar. If the subwindow is the bottom subwindow in the swmanager it will be sized to fill any remaining space in the swmanager. Fine Point: If the subwindows are vertically displayed, size represents the height of the subwindow. If the subwindows are horizontally displayed, size represents the width of the subwindow. In the initial implementation (4.x) only vertically displayed subwindows are supported.

vertScrollbar indicates whether or not a vertical scrollbar is desired.
horizScrollbar indicates whether or not a horizontalscrollbar is desired. (In Figure 1 the

message window at the top of the shell has only a vertical scrollbar. The form and lower subwindows have both horizontal and vertical scrollbars).

adjustable indicates if the subwindows size is to be adjustable.

Note that there is no "place" parameter. Subwindows will be placed in their parent (swmanager) in the order in which they are created. The first subwindow is placed at [0,0], the next one at firstSize + 1 and so on.

Fine Point: In each of the procs described here passing a NIL swmanager will return an "orphaned" subwindow. An orphan is a subwindow that has no parent and is not placed in a window tree. An orphan can be inserted at some later time using the procedures available in **SubwindowManager**.

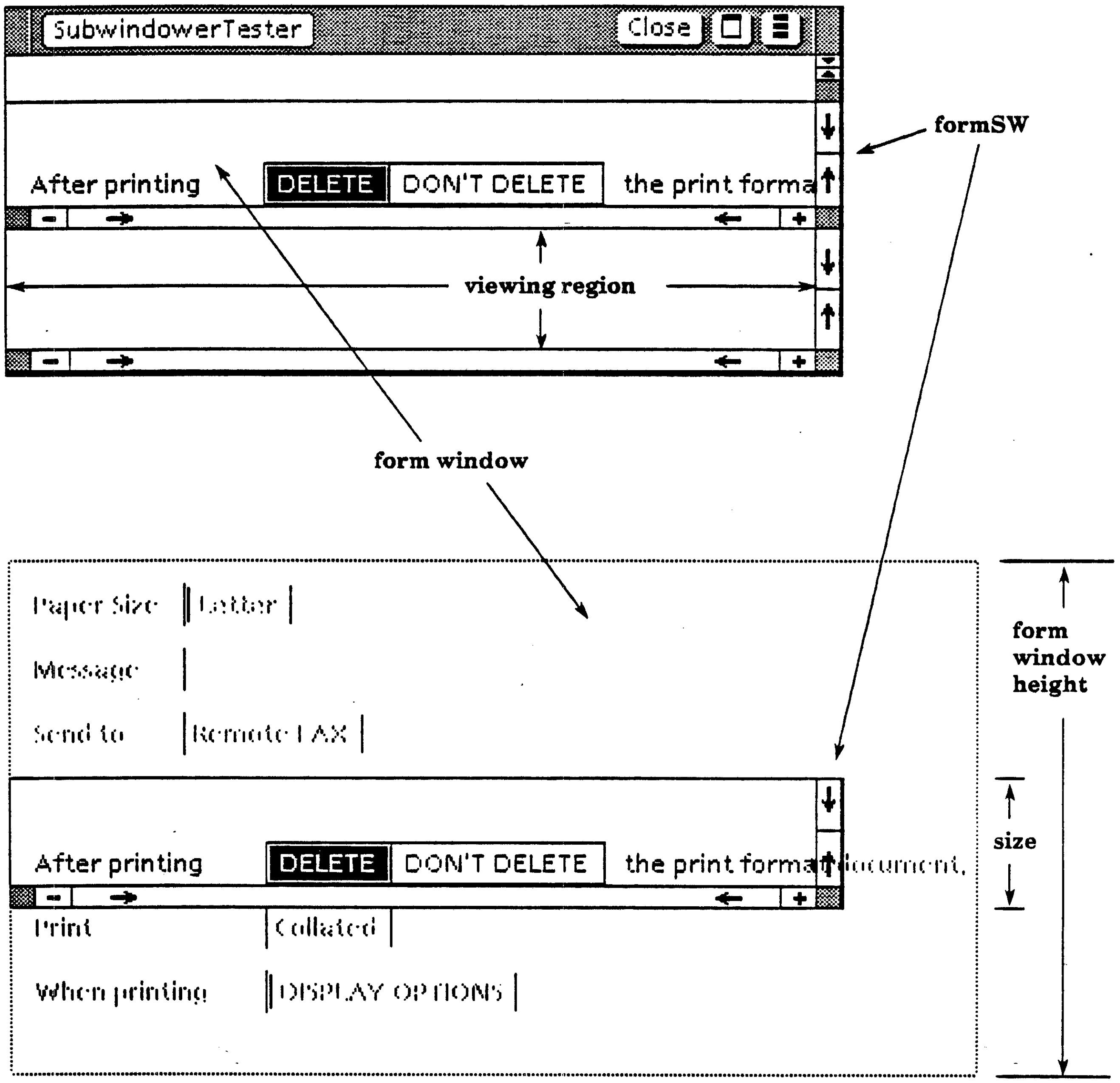
52.2.2 Creating Form subwindows

```
MakeFormSW: PROCEDURE[
    swmanager: Window.Handle,
    size: INTEGER← SubwindowManager.minSize,
    vertScrollbar, horizScrollbar, adjustable: BOOLEAN ← TRUE,
    makeItemsProc: FormWindow.MakeItemsProc,
    layoutProc: FormWindow.LayoutProc ← NIL,
              -- uses FormWindow.DefaultLayout if NIL
    windowChangeProc: FormWindow.GlobalChangeProc ← NIL,
    minDimsChangeProc: FormWindow.MinDimsChangeProc ← NIL,
              --uses FormWindow.DefaultMinDimsChangeProc if NIL
    zone: UNCOUNTED ZONE,
    clientData: LONG POINTER ← NIL]
    RETURNS [form: Window.Handle];
```

MakeFormSW creates a form subwindow. The client can specify if scrollbars are to be attached and if the window is to be adjustable. See §2.1 for specifics about **swmanager**, **size**, **vertScrollbar**, **horizScrollbar**, and **adjustable**.

makeItemsProc, **layoutProc**, **windowChangeProc**, **minDimsChangeProc**, **zone** and **clientData** are all identical to the parameters in **FormWindow.Create**. Refer to that chapter for further information.

The RETURN parameter **form** is the form window itself. Note that this form window is *inside* of the viewing region of the subwindow and is a *different* window from the formSW. (See figure 1 for a pictorial representation of the formSW and its descendant form window) The subwindow will be of size **size**. It's viewing region will be a scrollbar width smaller if scrollbars are present. The **form** window will be sized to "just fit" around all the items laid out within it by the **layoutProc**.



After printing DELETE DON'T DELETE the print format document.

Paper Size Letter

Message

Send to Remote FAX

Print Collated

When printing DISPLAY OPTIONS

Figure 1

Distinguishing the subwindow from its contents: The formSW contained in the shell in the first diagram is shown alone in the second diagram. The ghosted portions represent the hidden parts of the underlying form window. The formSW is shown to be sized such that only a small portion of its descendant form window is visible. The form is scrolled to show only the fourth item within it.

52.2.3 Creating Message subwindows

```
MakeMessageSW: PROCEDURE[
    swmanager: Window.Handle,
    size: INTEGER← SubwindowManager.minSize,
    vertScrollbar, horizScrollbar, adjustable: BOOLEAN ←TRUE,
    lines: CARDINAL ← 10,
    zone: UNCOUNTED ZONE]
    RETURNS [message: Window.Handle];
```

MakeMessageSW creates a message subwindow. The client can specify if scrollbars are to be attached or the window is to be adjustable. The effect of these parameters is described above in §2.1. The subwindow will be of size **size**. The height of the **message** window itself will be based on number of **lines**. (Note: A message subwindow with scrollbars will be a scrollbar width smaller than size to account for the addition of the scrollbars.) **lines** and **zone** are identical to the parameters in **MessageWindow.Create**. Refer to that chapter for further information.

52.2.4 Creating BodyWindowParent subwindows

```
MakeBodySW: PROC[
    swmanager: Window.Handle,
    size: INTEGER ← SubwindowManager.minSize,
    vertScrollbar, horizScrollbar, adjustable: BOOLEAN ←TRUE,
    adjustProc: AdjustableWindow.AdjustProc ←NIL,
    garbageCollectBodiesProc: BodyWindowParent.GarbageCollectBodiesProc ← NIL,
    moreScrollProc: BodyWindowParent.MoreScrollProc ←NIL,
    scrollbarInfoProc: Scrollbar.ScrollbarInfoProc ←BodyWindowParent.DefaultScrollbarInfo,
    thumbFeedbackProc: Scrollbar.ThumbFeedbackProc ← NIL,
    thumbScrollProc: Scrollbar.ThumbScrollProc ←BodyWindowParent.DefaultThumbScroll,
    zone: UNCOUNTED ZONE]
    RETURNS [bodyParent: Window.Handle];
```

MakeBodySW creates a **bodyParent** window that is a descendant of **swmanager**. **swmanager**, **vertScrollbar**, **horizScrollbar**, and **adjustable** are described in detail in §2.1 of this chapter. See the BodyWindowParent chapter for information about **adjustProc**, **scrollbarInfoProc**, **moreScrollProc**, **garbageCollectBodiesProc**, **thumbFeedbackProc**, **thumbScrollProc** and **zone**.

The RETURN parameter **bodyParent** is the bodySW itself NOT an individual "body" window. The client is expected to place a body or bodies inside the bodyParent window by calls to **BodyWindowParent.CreateBody**. For example, a bodyParent without bodies is uninteresting. Refer to the BodyWindowParent chapter for further information regarding the relationship between a bodyParent and body windows.

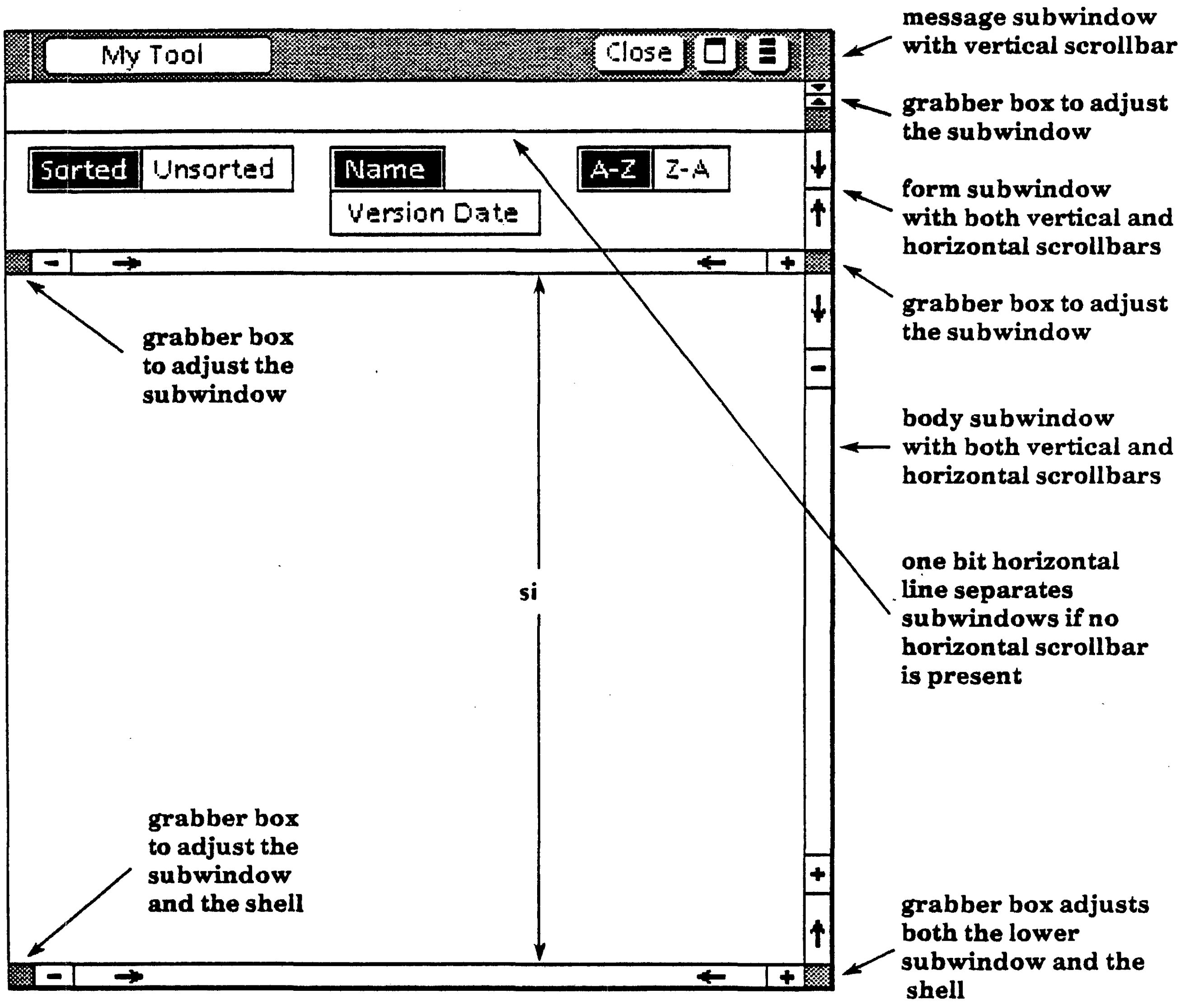

My Tool
Close
Sorted
Unsorted
Name
Version Date
A-Z
Z-A
si
message subwindow with vertical scrollbar
grabber box to adjust the subwindow
form subwindow with both vertical and horizontal scrollbars
grabber box to adjust the subwindow
grabber box to adjust the subwindow
body subwindow with both vertical and horizontal scrollbars
one bit horizontal line separates subwindows if no horizontal scrollbar is present
grabber box to adjust the subwindow and the shell
grabber box adjusts both the lower subwindow and the shell

Figure 2
Tool produced by the Sample Code in §3

52.3 Usage/Examples

```
CreateMyToolWindow: PROCEDURE = {
 myTool: XString.ReaderBody ← XString.FromSTRING["My Tool"L];
 shell: StarWindowShell.Handle ← StarWindowShell.Create [
     name: @myTool, scrollData: [FALSE,FALSE,NIL,NIL,NIL] ];
 z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
 swManager: Window.Handle ← StarWindowShellExtra5.ManagerFromShell[
     sws: shell];
 --establish some initial window heights
 shelldims: Window.Dims ← [375, 400]
 minimum: INTEGER ← 20;
 small: INTEGER ← 60;
 medium: INTEGER ← 300;
 bodyBox: Window.Box ← [[0,0],mumble];
 msgSW: Window.Handle ← Subwindower.MakeMessageSW[
     swmanager: swManager, size: minimum, horizScrollbar: FALSE, zone: z];
 formSW: Window.Handle ← Subwindower.MakeFormSW[
     swmanager:swManager, size: small, makeItemsProc: MakeItems, zone: z];
 bodySW: Window.Handle ← Subwindower.MakeBodySW[
     swmanager:swManager,size: medium, zone: z];
 body: Window.Handle ← BodyWindowParent.CreateBody[bodySW, bodyBox,
     DisplayProc, NotifyProc];
 StarWindowShell.SetPreferredInteriorDims[shell, shelldims];
 StarWindowShell.Push [shell];
 };

DisplayProc: PROCEDURE = {
 --how does my body window display itself?--
 };

NotifyProc: TIP.NotifyProc = {
 --what is my body window interested in seeing?--
 };

MakeItemsProc: FormWindow.MakeItemsProc = {
 --make some items using FormWindow.MakeXXXItem--
 };
```

52.4 Index of Interface Items

53

SubwindowFriends

53.1 Overview

SubwindowFriends provides a facility for registering new subwindow types. The client provides the various procedures that will be needed later when a subwindow of the new type is being created. For a comprehensive view of all the subwindow interfaces and their intended use, see the Subwindow Overview chapter.

53.2 Interface Items

53.2.1 Registering Subwindow types

```
UniqueSWType: PROCEDURE
   RETURNS [swType: SubwindowManager.SWType];
```

UniqueSWType registers a client-defined subwindow type returning a unique type that is used later by other procedures.

```
SetSWProcs: PROCEDURE [
   type: SubwindowManager.SWType,
   attachScrollbarsProc: AttachScrollbarsProc ← StandardAttachScrollbars,
   clientWindowFromSWProc: ClientWindowFromSWProc ←
      StandardClientWindowFromSW,
   adjustProc: AdjustableWindow.AdjustProc ← StandardAdjust,
   limitProc: AdjustableWindow.LimitProc ← StandardLimit,
   transitionProc: SubwindowManager.TransitionProc ← StandardTransition ];
```

Registers the procedures to be called by SubwindowManager.MakeSW (etc.) when creating, resizing or changing state of a sw of type.

attachScrollbarsProc is called when a window of type is being created. This proc is expected to call Scrollbar.Attach with the proper window etc.

clientWindowFromSWProc is called to get the client window when only the subwindow handle is known.

adjustProc is called when client subwindow of **type** is about to be resized (when = before) and again just after the resize occurs (when = after).

limitProc is called when client subwindow of **type** is about to be resized.

transitionProc is called when the state (**SubwindowManager.State**) of a subwindow of **type** is changing.

```
AttachScrollbarsProc: TYPE = PROCEDURE [
    subwindow: Window.Handle,
    vertScrollbar, horizScrollbar: BOOLEAN,
    zone: UNCOUNTED ZONE]
    RETURNS [sw: Window.Handle];
```

Client provides an **AttachScrollbarsProc** that will be called to attach scrollbars when **SubwindowManager.MakeSW** is called. **sw** may be a descendant of **subwindow** or the same as **subwindow**.

```
ClientWindowFromSWProc: TYPE = PROC [
    subwindow: Window.Handle]
    RETURNS [sw: Window.Handle];
```

Provides a means of getting the client window when only the subwindow handle is known.

53.2.2 Getting subwindow procs

```
GetSWProcs: PROCEDURE [
    type: SubwindowManager.SWType]
    RETURNS [
    attachScrollbarsProc: AttachScrollbarsProc,
    clientWindowFromSWProc: ClientWindowFromSWProc,
    adjustProc: AdjustableWindow.AdjustProc,
    limitProc: AdjustableWindow.LimitProc,
    transitionProc: SubwindowManager.TransitionProc ];
```

Returns the procedures registered with **type**.

53.2.3 Standard Procedures

```
StandardAttachScrollbars: AttachScrollbarsProc;
```

Creates a bodyParent assembly: namely a subwindow, descendant viewer and attached scrollbars. Adjust and scrolling behavior will be as for body windows. The subwindow returned is the window passed in.

```
StandardClientWindowFromSW: ClientWindowFromSWProc;
```

Returns the first descendant window encountered at the client level (i.e. below the bodyParent assembly).

StandardAdjust: AdjustableWindow.AdjustProc;

Adjusts the subwindow's descendant viewer and scrollbars.

StandardLimit:AdjustableWindow.LimitProc;

Prevents shrinking the subwindow below minSize.

StandardTransition: SubwindowManager.TransitionProc;

NoOp.

53.2.4 Errors

Error: ERROR [code: ErrorCode];

ErrorCode: TYPE = {noSuchType, tooManyTypes, other};

noSuchType	raised if **GetSWProcs** or **SetSWProcs** are called with an invalid type
tooManyTypes	raised if **UniqueSWType** has been called too many times
other	*not currently used*

53.3 Usage/Examples

```
 <<Simple code to create a tool with a new subwindow type>>
CreateMyToolWindow: PROCEDURE = {
    myTool: XString.ReaderBody ← XString.FromSTRING["My Tool"L];
    shell: StarWindowShell.Handle ← StarWindowShell.Create [
        name: @myTool, scrollData: [FALSE,FALSE,NIL,NIL,NIL] ];
    z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
    swManager: Window.Handle ← StarWindowShellExtra5.ManagerFromShell [
        sws: shell];
    --establish window heights
    medium: INTEGER ← 60;
    large: INTEGER ← 300;
    formSW: Window.Handle ← Subwindower.MakeFormSW[
        swmanager:swManager, size: medium] makeItemsProc: MakeItems, zone: z];
    mySW: Window.Handle ← SubwindowManager.MakeSW[
        swmanager:swManager, type: myType, size: large, zone: z];
    StarWindowShell.Push [shell];
    };
```

```
ScrollSW: SubwindowFriends.AttachScrollbarsProc = {
--call Scrollbar.Attach with the desired interior window etc.
};

MyWindow: SubwindowFriends.ClientWindowFromSWProc = {
--return the same window as ScrollSW given the same subwindow
};

AdjustProc : AdjustableWindow.AdjustProc = {
--adjusts itself and calls Scrollbar.Adjust with the proper interior window
};

LimitProc: AdjustableWindow.LimitProc = {
--impose any desired Limits
--prudent to call SubwindowFriends.StandardLimitProc to prevent shrinking
--below SubwindowManager.minSize.
};

TransitionProc: SubwindowManager.TransitionProc = {
--do whatever actions are specific to myType subwindow at various state changes
};

--Mainline Code

--create a unique type
myType: SubwindowManager.SWType ← SubwindowFriends.UniqueSWType[];
--Set the procs needed for myType subwindow (can default to Standard ones)
SubwindowFriends.SetSWProcs[type: myType,
    attachScrollbarsProc: ScrollSW, clientWindowFromSWProc: MyWindow,
    adjustProc: AdjustProc, limitProc: LimitProc, transitionProc: TransitionProc];
```

53.4 Index of Interface Items

Item	Page
AttachScrollbarsProc: TYPE	2
ClientWindowFromSWProc: TYPE	2
Error	2
ErrorCode: TYPE	2
GetSWProcs: PROCEDURE	2
SetSWProcs:PROCEDURE	1
StandardAttachScrollbars: AttachScrollbarsProc	2
StandardClientWindowFromSW: ClientWindowFromSWProc	2
StandardAdjust: AdjustableWindow.AdjustProc	2
StandardLimit:AdjustableWindow.LimitProc	2
StandardTransition: SubwindowManager.TransitionProc	2
UniqueSWType:PROCEDURE	1

54

SubwindowManager

54.1 Overview

SubwindowManager provides a facility for creating subwindows in a window. It is typically used in conjunction with **StarWindowShell.ManagerFromShell**. Fine point: in BWS 4.3, ManagerFromShell is in StarWindowShellExtra5. New types of subwindows may be defined using **SubwindowFriends** and then an instance of that subwindow created using **SubwindowManager**. Procedures for adding and removing subwindows from their parent are also provided. Fine point: Predefined types of subwindows (form, message, body) can be created using the procedures in **Subwindower** *instead.* See the Subwindow Overview Chapter for a more complete look at the appropriate use for the various subwindow interfaces.

54.2 Interface Items

54.2.1 Making Subwindows

```
MakeSW: PROCEDURE [
    swmanager: Window.Handle ← NIL,
    type: SWType ← vanilla,
    size: INTEGER ← minSize,
    vertScrollbar, horizScrollbar, adjustable: BOOLEAN ← TRUE,
    zone: UNCOUNTED ZONE ]
    RETURNS [sw: Window.Handle];
```

Creates a subwindow of **type** and **size**. **sw** is appended after the last subwindow in **swmanager**. While it is the bottom subwindow its size will be adjusted to fill the remaining space in its parent swmanager. If **swmanager** is NIL, **MakeSW** creates an orphan subwindow that can be added to a swmanager at a later time using **AppendSW** or **InsertSW** as described below in §2.2.

swmanager refers to the parent window that will contain the subwindow. (Note: a swmanager that manages the adjustment of adjacent subwindows may be created by calling StarWindowShellExtra5.ManagerFromShell.) If **swmanager** is not already a subwindow manager, it will become one.

type may be one of the predefinded types (**body, form, message** etc.) or a client defined type. Fine Point: see SubwindowFriends for creating client defined types. Currently there are accelerators available to create some of the predefined types. (See **Subwindower**) More will be implemented in the future. Most clients will use the accelerator procs to create these predefined types rather than MakeSW. MakeSW only creates the subwindow with appropriately attached scrollbars, it does NOT make the client call to FormWindow, MessageWindow, etc. See the Subwindower Chapter for more information.

size specifies the size of the subwindow (see Figure 1). If scrollbars are attached, the viewing region within the subwindow is reduced by the width of a vertical scrollbar. If the subwindow is the bottom subwindow in the swmanager it will be sized to fill any remaining space in the swmanager. Fine Point: If the subwindows are vertically displayed, size represents the height of the subwindow. If the subwindows are horizontally displayed, size represents the width of the subwindow. In the initial implementation (4.x) only vertically displayed subwindows are supported.

vertScrollbar indicates whether or not a vertical scrollbar is desired.

horizScrollbar indicates whether or not a horizontalscrollbar is desired. (In Figure 1 the message subwindow has only a vertical scrollbar while the other two subwindows have both vertical and horizontal scrollbars.)

adjustable indicates if the subwindow's size is to be adjustable.

Note that there is no "place" parameter. Subwindows will be placed in their parent (swmanager) in the order in which they are created. The first subwindow is placed at [0,0], the next one at firstSize + 1 and so on.

```
SWType: TYPE = MACHINE DEPENDENT {
    vanilla, body, form, message, container, log, text, list, last(377B)};
```

Enumerated for **SWType**. Fine Point: Four of the enumerated types of subwindows are currently implemented in 4.x. They are body, form, and message and vanilla. More of the predefined types are expected to be implemented later.

```
GetType: PROCEDURE [
    sw: Window.Handle]
    RETURNS [type: SWType];
```

Returns the **SWType** associated with sw. Can raise Error[notASubwindow].

```
minSize: INTEGER;
```

Minimum size for a subwindow. Used as the default for **size** in MakeSW.

54.2.2 Adding and removing subwindows

In the following procedures there is a common reference to **sw** and **swmanager**. These are described here rather than with each procedure. Each can raise Error[notASubwindow] or Error[notAswmanager]. The visual effects of any of these procedures will not take place until SubwindowManager.Repaint is called. **Repaint** is also described in this section.

sw refers to a subwindow that has been created using **MakeSW** described above or one of the **MakeXXXSW** procedures in Subwindower.

swmanager refers to the parent window that will contain the subwindow. (See §54.2.1).

```
AppendSW: PROC [
    sw, swmanager: Window.Handle,
    after: Window.Handle ← NIL];
```

Adds an existing **sw** to the **swmanager** after **after** (y= afterPlace.y+afterDims.h) or after the bottom subwindow in **swmanager** if **after = NIL**. Any subwindows below it will move down to allow space for the new subwindow. It is possible that they may become totally or partially obscured by the parent swmanager.

```
InsertSW: PROC [
    sw, swmanager: Window.Handle,
    before: Window.Handle ← NIL];
```

Adds **sw** to the **swmanager** before **before** or at the top (y=0) of the subwmanager if **before = NIL**. Any subwindows below it will move down to allow space for the new subwindow. It is possible that they may become totally or partially obscured by the parent swmanager.

```
RemoveSW: PROC [
    sw, swmanager: Window.Handle];
```

Removes **sw** from the tree of **swmanager**. Adjusts the location of any other subwindows in the **swmanager** as necessary to refill the space left by **sw**.

```
SwapSW: PROC [
    oldSW, newSW: Window.Handle];
```

Removes **oldSW** from its parent inserting **newSW** in its place in the tree. **newSW** will occupy the same location and size as **oldSW**.

```
ResizeSW:PROCEDURE [
    sw: Window.Handle,
    size: INTEGER]
    RETURNS [Window.Dims];
```

Allows the client to programmatically change the size of a subwindow. Returns the actual size used for the resize after the appropriate limitProcs are called. This proc is particularly useful to the client who wishes to redistribute the space when adding or subtracting subwindows.

```
Repaint: PROCEDURE [
    swmanager: Window.Handle];
```

If the **swmanager** is displayed, **Repaint** MUST be called in conjunction with any of the previous procedures: **AppendSW**, **InsertSW**, **RemoveSW**, **SwapSW**, and **ResizeSW**. This allows the client to make several calls to resize, add or delete subwindows before validating the changes.

54.2.3 Adjust, Limit, Transition Types and Procs

State: TYPE = MACHINE DEPENDENT {awake(0), sleeping, dead, last(7)};

A window is always in one of three states:

awake:	displayed
sleeping:	created but not displayed
dead:	being destroyed

TransitionProc: TYPE = PROC [window: Window.Handle, state: State];

A **TransitionProc** is a client-supplied procedure responsible for allocating or deallocating client data structures when the **window's** state changes. **state** is the new state of the window.

Please note that most clients will have no need for the next 3 procedures.

Transition: TransitionProc;

Transition is the TransistionProc associated with the SubwindowManager window.

Adjust: AdjustableWindow.AdjustProc;

An AdjustProc is a client-supplied procedure that is called every time the window is changing size. **Adjust** is the AdjustProc associated with the SubwindowManager window. It is to be called whenever the manager is adjusted. For example, the StarWindowShell adjust code calls SubwindowManager.Adjust[shellsSWmanager] when the shell is being resized.

Limit: AdjustableWindow.LimitProc;

A LimitProc is a client-supplied procedure that is called every time the window is about to change size or location. This gives the client a chance to veto or change the new box before the adjustment actually takes place. **Limit** is the LimitProc associated with the SubwindowManager window.

54.2.4 Utilities

```
EnumerateSWs: PROCEDURE [
    swmanager: Window.Handle,
    proc: EnumSWsProc]
    RETURNS [sw: Window.Handle];
```

EnumSWsProc: TYPE = PROCEDURE [sw: Window.Handle] RETURNS [stop: BOOLEAN ← FALSE] ;

Call **proc** with each **sw** contained in **swmanager**. Can raise Error[notAswmanager]. The client may make calls to add/delete subwindows during the course of the enumeration without adverse effects.

GetZone: PROCEDURE [sw: Window.Handle] RETURNS [zone: UNCOUNTED ZONE];

Returns the **zone** associated with the subwindow (**sw**).

```
IsItManager: PROCEDURE [window: Window.Handle] RETURNS [BOOLEAN];
```

Returns TRUE if the **window** is a subwindow manager.

```
IsItSubwindow: PROCEDURE [window: Window.Handle] RETURNS [BOOLEAN];
```

Returns TRUE if the **window** is a subwindow.

```
ManagerFromSW: PROCEDURE [
   sw: Window.Handle]
   RETURNS [swmanager: Window.Handle];
```

Return the **swmanager** containing **sw**. Returns NIL if **sw** is orphaned.

```
SubwindowFromChild: PROCEDURE [
   window: Window.Handle]
   RETURNS [Window.Handle ← NIL];
```

Return the subwindow containing **window**. Returns NIL if **window** is not a child of a subwindow.

54.2.5 Errors

```
Error: ERROR [code: ErrorCode];

ErrorCode: TYPE = {notASubwindow, notAswmanager};
```

notASubwindow	raised if **sw** is not a subwindow
notAswmanager	raised if **swmanager** is not a swmanager

54.3 Usage/Examples

<<This code will create a window shell with a message subwindow, a form subwindow and a vanilla subwindow. It will be up to the client to then "do something interesting" with the vanilla subwindow. See Figure 1 for a pictorial view of the tool created by this code.>>

```
CreateMyToolWindow: PROCEDURE = {
 myTool: XString.ReaderBody ← XString.FromSTRING["My Tool"L];
 shell: StarWindowShell.Handle ← StarWindowShell.Create [
    name: @myTool, scrollData: [FALSE,FALSE,NIL,NIL,NIL] ];
 z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
 swManager: Window.Handle ← StarWindowShellExtra5.ManagerFromShell [
    sws: shell];
 --establish initial window heights
 small: INTEGER ← 20;
 medium: INTEGER ← 60;
 large: INTEGER ← 300;
 msgSW: Window.Handle ← Subwindower.MakeMessageSW[
    swmanager:swManager, size: small] horizScrollbar: FALSE, zone: z];
 formSW: Window.Handle ← Subwindower.MakeFormSW[
      swmanager:swManager, size: medium, makeItemsProc: MakeItems, zone: z];
  mySW: Window.Handle ← SubwindowManager.MakeSW[
      swmanager:swManager,size: large, zone: z];
  StarWindowShell.Push [shell];
};
MakeItemsProc: FormWindow.MakeItemsProc = {
  --make some items using FormWindow.MakeXXXItem--
  };
```

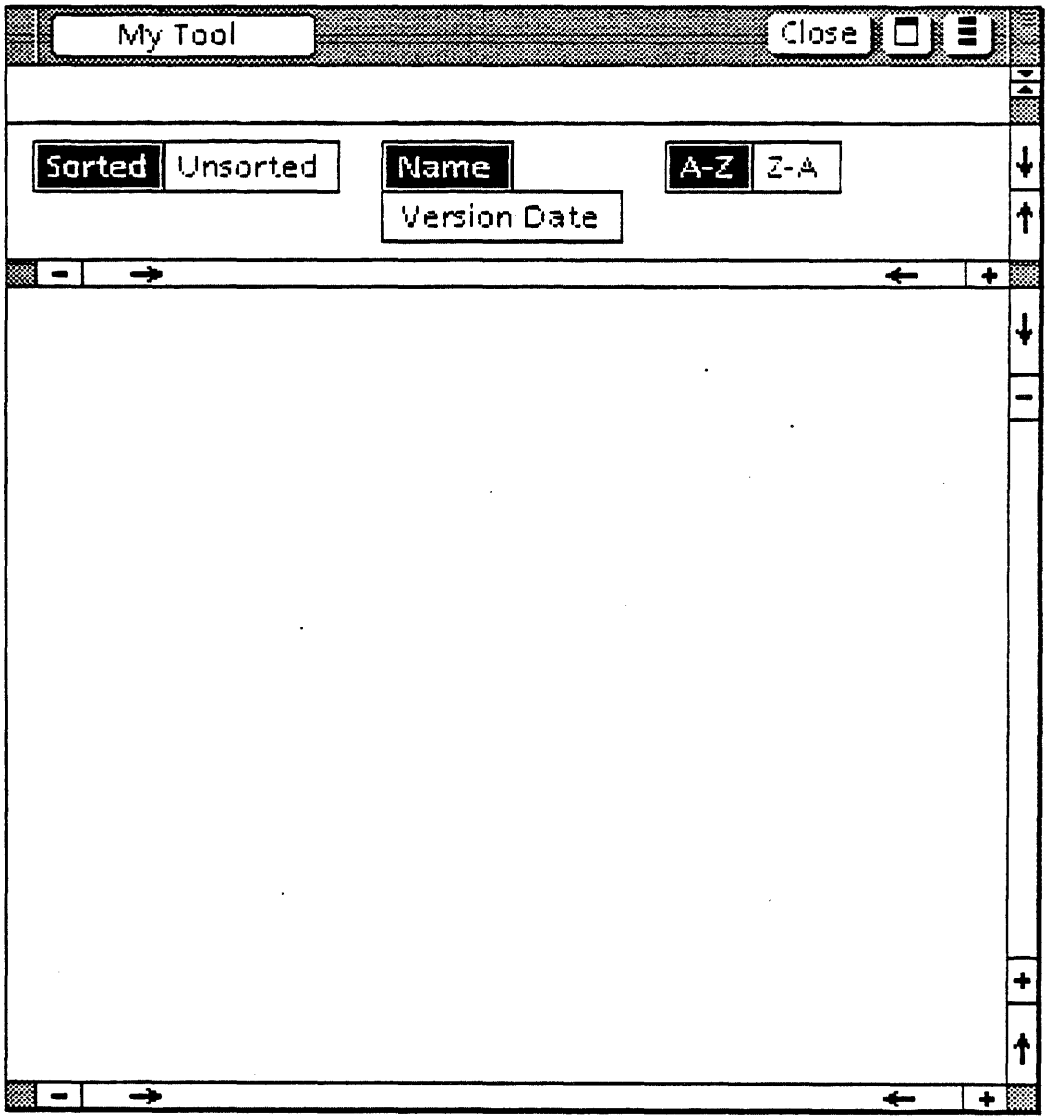


Figure 1. Tool created in sample code

<<The following code segments will create a window shell with a form subwindow and a body subwindow. The header will have and extra menu item "AddFoo" for the user to select. Selecting "AddFoo" will cause another form subwindow to be added at the top of the subwindows. The "AddFoo" command will change to "RemoveFoo" at that time. Selecting "RemoveFoo" will remove the new subwindow from the shell and exchange the commands in the header.>>

```
fooSW: Window.Handle ← NIL;
addItem, removeItem: MenuData.ItemHandle;

CreateMyToolWindow: PROCEDURE = {
  myTool: XString.ReaderBody ← XString.FromSTRING["My Tool"L];
  addOn: XString.ReaderBody ← XString.FromSTRING["Add Foo"L];
  remove: XString.ReaderBody ← XString.FromSTRING["Remove Foo"L];
  shell: StarWindowShell.Handle ← StarWindowShell.Create [
      name: @myTool, scrollData: [FALSE,FALSE,NIL,NIL,NIL] ];
  z: UNCOUNTED ZONE ← StarWindowShell.GetZone [shell];
  removeItem ← MenuData.CreateItem [zone: z, name: @remove, proc: RemoveProc];
  addItem ← MenuData.CreateItem [zone: z, name: @addOn, proc: AddOnProc];
  item: ARRAY [0..1) OF MenuData.ItemHandle ← [addItem];
  myMenu: MenuData.MenuHandle = MenuData.CreateMenu [
      zone: z, title: NIL, array: DESCRIPTOR [item]];
  swManager: Window.Handle ← StarWindowShellExtra5.ManagerFromShell [
      sws: shell];
  --establish initial window heights
  medium: INTEGER ← 60;
  large: INTEGER ← 300;
  formSW: Window.Handle  ← Subwindower.MakeFormSW[
      swmanager:swManager, size: medium] makeItemsProc: MakeItems, zone: z];
  bodySW: Window.Handle  ← Subwindower.MakeBodySW[
      swmanager:swManager, size:large] zone: z];
  body: Window.Handle ← BodyWindowParent.CreateBody[
      bodySW, displayProc: DisplayProc, notify: NotifyProc];
  --create an orphan FormSW of minimum dimensions
  fooSW ← Subwindower.MakeFormSW[
      makeItemsProc: MakeNewItems, zone:z];
  StarWindowShell.SetRegularCommands [sws: shell, commands: myMenu];
  StarWindowShell.Push [shell] };

AddOnProc: MenuData.MenuProc = {
  --add my type subwindow at the top of the shell
  SubwindowManager.InsertSW[fooSW, swManager];
  --swap the menu items
  MenuData.SwapItem[old:addItem, new: RemoveItem] };

RemoveProc: MenuData.MenuProc = {
  --remove my type subwindow from the top of the shell
  SubwindowManager.RemoveSW[fooSW, swManager];
  --swap the menu items
  MenuData.SwapItem[old: removeItem, new: addItem] };
```

54.4 Index of Interface Items

Item	Page
Adjust: AdjustableWindow.AdjustProc	4
AppendSW:PROCEDURE	3
EnumerateSWs:PROCEDURE	4
EnumSWSProc: TYPE	4
Error	5
ErrorCode: TYPE	5
GetType: PROCEDURE	2
GetZone: PROCEDURE	4
InsertSW:PROCEDURE	3
IsItManager:PROCEDURE	5
IsItSubwindow:PROCEDURE	5
Limit: AdjustableWindow.LimitProc	4
MakeSW:PROCEDURE	1
ManagerFromSW:PROCEDURE	5
minSize:INTEGER	2
RemoveSW:PROCEDURE	3
Repaint:PROCEDURE	3
ResizeSW:PROCEDURE	3
State:TYPE	4
SubwindowFromChild: PROCEDURE	5
SwapSW: PROCEDURE	3
SWType:TYPE	2
Transition: TransitionProc	4
TransitionProc: TYPE = PROCEDURE	4

55

TIP

55.1 Overview

TIP provides basic user input facilities through a flexible mechanism that translates hardware-level actions from the keyboard and mouse into higher-level client action requests (result lists). The acronym TIP stands for *terminal interface package*. This interface also provides the client with routines that manage the input focus, the periodic notifier, and the **STOP** key.

55.1.1 Basic Notification Mechanism

The basic notification mechanism directs user input to one of many windows in the window tree. Each window has a **Table** and a **NotifyProc**. The table is a structure that translates a sequence of user actions into a sequence of results that are then passed to the notify procedure of the window.

There are two processes that share the notification responsibilities, the Stimulus process and the Notifier process. The Stimulus process is a high-priority process that wakes up approximately 50 times a second. When it runs, it makes the cursor follow the mouse and watches for keyboard keys going up or down, mouse motion, and mouse buttons going up or down, enqueuing these events for the Notifier process.

The Notifier process dequeues these events, determines which window the event is for, and tries to match the events in the window's table. If it finds a match in the table, it calls the window's notify procedure with the results specified in the table. If no match is found, it tries the next table in the window's chain of tables. If no match is found in any table, the event is discarded.

The Notifier process is an important process. To avoid multi-process interference, some operations in the system are restricted to happen only in the Notifier process. Setting the selection is one such operation. The Notifier process is also the one most closely tied to the user. If an operation will take an extended time to complete, it should be forked from the notifier process to run in a separate process so that the Notifier process is free to respond to the user's actions.

55.1.2 Tables

Tables provide a flexible method of translating user actions into higher-level client-defined actions. They are essentially large select statements in which the user's actions are matched against the left side of a table with a corresponding set of results on the right side. Left sides of tables specify *triggers*--changes in state of keys--and *enablers*--existing state of keys--to be matched with the user actions. Right sides of tables specify results that include mouse coordinates, atoms, and strings for keyboard character input. A complete syntax and semantics of tables are in §55.3.2 and §55.3.3.

Tables have a user-editable filed representation that is parsed to build a runtime data structure for **TIP**. The user-editable filed representation must be in the system catalog and is assumed to have a file name extension of .TIP. When a table is created by calling the **CreateTable** operation, the tip file is parsed and its runtime data structure is created. In addition, a compiled version of the tip file is created in a file whose name has the character C appended to the name of the tip file. On subsequent calls to **CreateTable**, this tipc file is used to create the runtime data structure as long as the tip file has not been changed. By avoiding parsing the tip file, building the runtime data structure from the compiled file is much faster.

The table parser uses a macro package that allows macros to be defined and used in writing tables. It is described in §55.3.7.

Tables may be linked to form a chain of tables. The notifier process attempts to match user actions in the first table of the chain. If no match is found, it tries subsequent tables in the chain. If no match is found in any table, the user action is discarded. Clients can use the links of tables to build special processing on top of basic facilities. The client can write its own table to handle special user actions and, by linking the table to system-defined tables, let them handle the normal user actions. An example is in the Usage/Examples section. System-defined tables, which are accessable through the **TIPStar** interface, are described in **Appendix A**.

55.1.3 Input Focus

The input focus is a distinguished window that is the destination of most user actions. User actions may be directed either to the window with the cursor or to the input focus. Actions such as mouse buttons are typically sent to the window with the cursor. Most other actions, such as keystrokes, are usually sent to the current input focus.

The notifier process uses either the input focus window or the cursor window to obtain a table and a notify procedure. Results from matching the user actions with the table are normally passed to the window's notify procedure. Notify procedures may also be bound directly with a table. In this case, results from the table go to the table's notify procedure instead of to the window's notify procedure.

Clients may make a window be the current input focus and be notified when some other window becomes the current input focus.

55.1.4 Periodic Notification

The Notifier process is important because it responds directly to the user. To avoid multi-process interference, some operations in the system are restricted to happen only in the Notifier process. The periodic notification mechanism allows operations to happen within the Notifier process while the user is idle. A periodic notifier is created with a window, some results, and a time interval. The window's notify procedure is called with the results every time interval as long as the Notifier process is not processing user actions.

55.1.5 Call-Back Notification and Setting the Manager

Call-back notification and setting of the manager bypass the normal means of selecting a window as the destination of input and allow the client to receive all input. It is useful for something like mouse tracking when a menu is posted. This must be done only from the Notifier process.

Call-back notification is so named because the client calls back to the Notifier process with a special call-back notification procedure, a window and a table. The Notifier matches user actions with the table and sends the results to the call-back notify procedure. It continues to do this until the call-back notify procedure says it is done. User actions that are not matched are discarded.

Setting the manager makes the Notifier process use the manager's table for matching user actions and send the results to the manager's notify procedure. User actions that are not matched are discarded.

While both call-back notification and setting the manager results in all notifications being sent to a single place, they are different in the control structure. With call-back notification, the client's call stack is not unwound, while setting the manager does not take effect until the current notification is processed and its call stack unwound.

55.1.6 Attention and User Abort

While most notifications are sent to notify procedures from the notifier process, there is a mechanism that allows asynchronous notification of the STOP key. An **AttentionProc** may be set for a window that is called whenever the STOP key is depressed in that window. It is called from outside the notifier process as soon as the stimulus level sees the key go down. For those windows that do not set an **AttentionProc**, the system keeps a user abort flag that records whether the STOP key was depressed. Clients may call **UserAbort** to check if the flag is set. It is cleared when any notification is sent to the window's notify procedure.

55.1.7 Stuffing Input into a Window

TIP provides operations that allow a client program to call the notify procedure of a window with results that the client constructs. **StuffResults** allows a client to pass an arbitrary results list to a window. **StuffString**, **StuffSTRING**, and **StuffCharacter** allow strings and characters to be passed to a window.

55.2 Interface Items

55.2.1 Results

```
Results: TYPE = LONG POINTER TO ResultObject;

ResultObject: TYPE = RECORD [
  next: Results,
  body: SELECT type: * FROM
    atom => [a: ATOM],
    bufferedChar => NULL,
    coords => [place: Window.Place],
    int => [i: LONG INTEGER],
    key => [key: KeyName, downUp: DownUp],
    nop => [],
    string => [rb: XString.ReaderBody],
    time => [time: System.Pulses],
    ENDCASE];

ATOM: TYPE = Atom.ATOM;

DownUp: TYPE = LevelIVKeys.DownUp; -- {down, up}

KeyName: TYPE = LevelIVKeys.KeyName;
```

The right side of a statement in a table is a list of results to be passed to the notify procedure when there is a match on the left side. Each element of this list of results is described by a **ResultObject**. The **atom** variant contains the atom from the table's right side. The place in the **coords** variant is relative to the window receiving the results. The reader body of the **string** variant is either from the **bufferedChar** results or from a string constant in the table. The pulses of the time variant is the value of System.GetPulses at the time the event actually occurred.

Character input is buffered by the Notifier process. If the result of a match is a bufferedChar, the Notifier process buffers the character and proceeds to try and match the next user actions. If there are no more user actions and the buffer of character input is not empty, the Notifier process calls the notify procedure with the buffered character input. If the next action produces a result that is not another character input, the Notifier process calls the notify procedure with the buffered character input and then call it with the new result. If the notifier process gets behind the user and a lot of character input actions get queued up, it collects them and passes them together to the client instead of one at at time.

KeyName is an enumerated that describes the keyboard and mouse buttons. See the *Pilot Programmer's Manual* (610E00160) for a complete list of the elements.

55.2.2 Notify Procedure

```
NotifyProc: TYPE = PROCEDURE [window: Window.Handle, results: Results];
```

A **NotifyProc** is the means of notifying a window of user input. The parameters are the window that is receiving the input and the list of results that describe the input. Normally, the results are from the tip table associated with the window. Notify procedures may also be bound directly with a table. In this case, results from the table go to the table's notify procedure instead of to the window's notify procedure.

55.2.3 TIP Tables

Table: TYPE = LONG POINTER TO **TableObject;**

TableObject: TYPE;

Table is a pointer to the internal representation of a table.

GetTableLink: PROCEDURE **[from: Table]** RETURNS **[to: Table];**

SetTableLink: PROCEDURE **[from, to: Table]** RETURNS **[old: Table];**

GetTableLink and **SetTableLink** allow the tables to be linked. **GetTableLink** returns the table following **from** in the chain, returning NIL if there is no successor table. **SetTableLink** sets the link of table **from** to **to** and returns the old value.

SetTableOpacity: PROCEDURE **[table: Table, opaque:** BOOLEAN**]**
  RETURNS **[oldOpaque:** BOOLEAN**];**

GetTableOpacity: PROCEDURE **[table: Table]** RETURNS **[**BOOLEAN**];**

SetTableOpacity sets the opacity of **table** and returns the old value, while **GetTableOpacity** returns its value. If a table is opaque, then unrecognized user actions are discarded without searching the table chain past the opaque entry.

55.2.4 Associating Notify Procedures, Tables, and Windows

SetTableAndNotifyProc: PROCEDURE **[**
  window: Window.Handle, table: Table ← NIL, **notify: NotifyProc ←** NIL**];**

SetTableAndNotifyProc makes window a TIP client and associates the table and notify procedure with the window. If **window** is already a TIP client and **table** or **notify** is NIL, then the old value is retained.

SetTable: PROCEDURE **[window: Window.Handle, table: Table]** RETURNS **[oldTable: Table];**

GetTable: PROCEDURE **[window: Window.Handle]** RETURNS **[Table];**

SetTable sets the table associated with **window** to be **table** and returns the old table. **GetTable** returns the table associated with **window**.

SetNotifyProc: PROCEDURE **[window: Window.Handle, notify: NotifyProc]**
  RETURNS **[oldNotify:NotifyProc];**

GetNotifyProc: PROCEDURE **[window: Window.Handle]** RETURNS **[NotifyProc];**

SetNotifyProc sets the notify procedure associated with **window** to be **notify** and returns the old notify procedure. **GetNotifyProc** returns the notify procedure associated with **window**.

SetNotifyProcForTable: PROCEDURE [table: Table, notify: NotifyProc]
RETURNS [oldNotify: NotifyProc];

GetNotifyProcFromTable: PROCEDURE [table: Table] RETURNS [NotifyProc];

SetNotifyProcForTable binds the notify procedure, **notify**, with **table** and returns the old value of bound notify procedure. Results from matches within the table will go to **notify** instead of to the notify procedure for the window this table is associated with. **GetNotifyProcFromTable** returns the notify procedure bound to **table**.

55.2.5 Creating and Destroying Tables

CreateTable: PROCEDURE [
file: XString.Reader, z: UNCOUNTED ZONE ← NIL, contents: XString.Reader ← NIL]
RETURNS [table: Table];

CreateTable generates a TIP table from the text file named by **file** (which may not be NIL). **file** is expected to be in the system file catalog. Storage for the table will be allocated in **z** or from the implementation's zone if **z** is NIL. **contents** is the default contents of **file**, and will be used if (1) the 'I boot switch is set, (2) the **file** cannot be read, or (3) the signal **InvalidTable** is resumed. (See **InvalidTable** for further details on how to treat that SIGNAL.)

When **file** is parsed, a compiled form of the table is written into a file with a name constructed by appending a C on the end of **file**. Fine Point: **file** should typically have the extension .TIP. When **CreateTable** is called, if a .TIPC file exists that was created from **file**, the .TIPC file is used to generate **table**. If the 'O boot switch is set, **CreateTable** does not look for a .TIP file, but rather looks directly for a .TIPC file. May raise the SIGNAL **InvalidTable**.

CreateCharTable: PROCEDURE [
z: UNCOUNTED ZONE ← NIL, buffered: BOOLEAN ← TRUE] RETURNS [table: Table];

CreateCharTable creates a TIP table such that any down transition of any of the keystations (not key tops) in the main typing array have a right-hand side of BUFFEREDCHAR. Storage is allocated from **z** if it is non-NIL or from the TIP implementation's zone. The boolean **buffered** is ignored and will be removed when this interface is updated.

CreatePlaceHolderTable: PROCEDURE [z: UNCOUNTED ZONE ← NIL] RETURNS [table: Table];

CreatePlaceHolderTable creates a placeholder tip table. Placeholder tables contain no information themselves but allow other tables with information to be linked to them. Storage is allocated from **z** if it is non-NIL or from the TIP implementation's private zone.

DestroyTable: PROCEDURE [LONG POINTER TO Table];

DestroyTable frees the table addressed by the parameter and then sets the table to NIL.

55.2.6 Input Focus

```
SetInputFocus: PROC [
   w: Window.Handle, takesInput: BOOLEAN, newInputFocus: LosingFocusProc ← NIL,
   clientData: LONG POINTER ← NIL];
```

SetInputFocus makes the window **w** the input focus. If **w** allows type-in, **takesInput** should be set to TRUE; otherwise **takesInput** should be set to FALSE. **newInputFocus** is called when **w** loses the input focus. It is passed **clientData** as the value of its LONG POINTER parameter. If **w** = NIL, the input focus is cleared, i.e. input focus notifications are sent to the **backStopInputFocus** (see below).

```
LosingFocusProc: TYPE = PROCEDURE [w: Window.Handle, data: LONG POINTER];
```

LosingFocusProc describes the procedure type that is used to let the input focus know it is no longer the input focus. **w** is the **Window.Handle** of the window that was the input focus, and **data** is the client data passed to **SetInputFocus**.

```
GetInputFocus: PROC RETURNS [Window.Handle];
```

GetInputFocus returns the window that currently has the input focus.

```
backStopInputFocus: READONLY Window.Handle;
```

The window **backStopInputFocus** gets all input focus notification when no other window requests to be the input focus. It may be NIL.

```
SetBackStopInputFocus: PROCEDURE [window: Window.Handle];
```

SetBackStopInputFocus sets **backStopInputFocus**, the window that gets all input focus notification if no other window requests to be the input focus.

```
FocusTakesInput: PROC RETURNS [BOOLEAN];
```

FocusTakesInput returns TRUE if the current input focus accepts input, FALSE otherwise.

```
ClearInputFocusOnMatch: PROC [w: Window.Handle];
```

ClearInputFocusOnMatch is used to clear the input focus in a window if that window has the input focus. This procedure is usually called by clients who are implementing their own window type when they are destroying a window.

55.2.7 Character Translation

```
CharTranslator: TYPE = RECORD [
   proc: KeyToCharProc, data: LONG POINTER];

KeyToCharProc: TYPE = PROCEDURE [
   keys: LONG POINTER TO KeyBits, key: KeyName, downUp: DownUp, data: LONG POINTER,
   buffer:XString.Writer];
```

A **CharTranslator** is used to construct characters from the state of the keyboard when a BUFFEREDCHAR result is encountered. The **KeyToCharProc** is called when the notifier needs to construct a character from the state of the keyboard. **keys** describes the current state of the keyboard. **key** and **downUp** describe the current character transition. The procedure should append the corresponding character(s) to **buffer**. There is a **CharTranslator** for each table.

KeyBits: TYPE = LevelIVKeys.KeyBits;

SetCharTranslator: PROCEDURE [table: Table, new: CharTranslator]
  RETURNS [old: CharTranslator];

GetCharTranslator: PROC [table: Table] RETURNS [o: CharTranslator];

SetCharTranslator sets the character translator for **table**, returning the old value. **GetCharTranslator** returns the character translator for **table**.

55.2.8 Periodic Notification

PeriodicNotify: TYPE [1];

nullPeriodicNotify: PeriodicNotify;

PeriodicNotify is a handle for a periodic notifier. Periodic notifiers are a means of notifying windows at regular intervals from within the Notifier. **nullPeriodicNotify** is the null value for **PeriodicNotify**.

CreatePeriodicNotify: PROC [
  window: Window.Handle, results: Results, milliSeconds: CARDINAL,
  notifyProc: NotifyProc ← NIL]
  RETURNS [PeriodicNotify];

CreatePeriodicNotify registers a periodic notification. If **notifyProc = NIL** then the notify proc associated with window is used. If **notifyProc # NIL**, is called; this is useful if the client is running in a background process but wants to perform some operation that must be done in the notifier process, such as obtaining the current selection. If there is a **COORDS** result and **window = NIL** , that result is [0, 0]. If **notifyProc = NIL** and **window = NIL** , the **Error[other]** is raised. The specified notify proc is called with parameters **window** and **results** once every **milliSeconds** milliseconds as long as no user action notifications are taking place. If **milliSeconds** = 0, it runs once and then destroys itself. The result list should not contain any entries of type **nop** or **bufferredChar**. Right-hand sides of type **coords** will be adjusted to reflect the actual mouse position relative to the window being notified. The results list will *not* be copied. Its allocation is up to the client.

CancelPeriodicNotify: PROC [
  PeriodicNotify] RETURNS [null: PeriodicNotify];

CancelPeriodicNotify stops the periodic notification passed in by removing the notification from the list of registered procedures and returns **nullPeriodicNotify**. This procedure raises **Error[noSuchPeriodicNotifier]** if the handle passed in is not valid (calling it with **nullPeriodicNotify** has no effect).

55.2.9 Call-Back Notification

CallBack: PROCEDURE [window: Window.Handle, table: Table, notify: CallBackNotifyProc];

CallBackNotifyProc: TYPE = PROCEDURE [window: Window.Handle, results: Results] RETURNS [done: BOOLEAN];

Call-back notification allows the client to receive all input. It is useful for something like mouse tracking when a menu is posted. **CallBack** uses **table** to match all user input and calls **notify** for each successful match in the table with **window** and the results from the table as parameters. **CallBack** will continue to send all notifications to **notify** as long as **notify** returns **done: FALSE**. Call-back notification is similar to setting the **Manager** except that the client's call stack is not unwound. User actions that are not matched are discarded.

55.2.10 Manager

Manager: TYPE = RECORD [
table: Table, window: Window.Handle, notify: NotifyProc];

nullManager: Manager = [NIL, TRASH, TRASH];

Manager is used to send all user actions through **table** and **notify**, using **window** instead of through the window, table, and notify procedure determined by **actionToWindow** and **TIPs** Match process. If **table** is **NIL**, as in **nullManager**, then the standard mechanisms will be used to determine where actions will be sent.

GetManager: PROC RETURNS [current: Manager];

GetManager returns the current manager.

ClearManager: PROCEDURE = INLINE ...;

ClearManager sets the manager to **nullManager**. It should only be called by clients who know that they set the manager from null to non-null.

SetManager: PROCEDURE [new: Manager] RETURNS [old: Manager];

SetManager does the obvious thing.

55.2.11 User Abort

UserAbort: PROC [Window.Handle] RETURNS [BOOLEAN];

UserAbort returns whether the user abort flag is set for the window. The bit may be set by calling **SetUserAbort** or by the system if the window does not have an attention procedure and the **STOP** key is depressed in that window. If **window** is **NIL**, the **UserInput** package checks to see whether the user has done a global abort. When a non-shift key goes down, this flag and the global abort flag are cleared.

ResetUserAbort: PROC [Window.Handle];

ResetUserAbort sets user abort flag for the window to **FALSE**.

SetUserAbort: PROC [Window.Handle]

SetUserAbort sets the user abort flag for the window. This does not call the window's attention procedure, if there is one.

55.2.12 Attention

AttentionProc: TYPE = PROC [window: Window.Handle];

An **AttentionProc** is a procedure called whenever the **STOP** key is depressed. It is called from a high-priority process—not the Notifier—as soon as the stimulus level sees the key go down.

SetAttention: PROC [Window.Handle, attention: AttentionProc]

SetAttention sets the attention procedure for the window. The procedure **attention** is called asynchronously whenever the **STOP** key is depressed.

55.2.13 Stuffing Input into a Window

StuffCharacter: PROC [
window: Window.Handle, char: XString.Character] RETURNS [BOOLEAN]

StuffCharacter allows a client to drive the type-in mechanism as though a character were coming from the user. The notify procedure of **window** is called with a string result that contains **char**. The returned **BOOLEAN** is **TRUE** only if **window** was prepared to accept input.

StuffCurrentSelection: PROC [window: Window.Handle] RETURNS [BOOLEAN]

StuffCurrentSelection allows a client to drive the type-in mechanism as though the contents of the current selection were coming from the user. The selection is converted to a string and the string is passed to the window's notify proc. (See the **Selection** interface for a description of the current selection.) The returned **BOOLEAN** is **TRUE** only if **window** was prepared to accept input.

StuffResults: PROCEDURE [window: Window.Handle, results: Results];

StuffResults calls the notify proc of **window** with **results**.

StuffString: PROC [window: Window.Handle, string: XString.Reader] RETURNS [BOOLEAN]

StuffString allows a client to drive the type-in mechanism as though **string** were coming from the user. The notify procedure of **window** is called with a string result that contains **string↑**. The returned **BOOLEAN** is **TRUE** only if **window** was prepared to accept input.

StuffSTRING: PROCEDURE [window: Window.Handle, string: LONG STRING]
RETURNS [BOOLEAN];

StuffSTRING calls the notify procedure of **window** with a results list that contains a **string ResultObject** whose reader body describes the characters in **string**.

StuffTrashBin: PROC [window: Window.Handle] RETURNS [BOOLEAN]

StuffTrashBin is currently not implemented.

55.2.14 Errors

InvalidTable: SIGNAL [type: TableError, message: XString.Reader];

TableError: TYPE = {fileNotFound, badSyntax};.

InvalidTable is only raised by **CreateTable**. The type is **fileNotFound** if the file could not be found and the **message** string was empty. **fileNotFound** is raised as an ERROR . The type is **badSyntax** if the current file is syntactically incorrect. If **badSyntax** is RESUMEd, and **message** is not empty, the **message** is written into **file**, and it is reparsed. If the file has been overwritten, or **message** is empty, and there is a syntax error, the error will be **badSyntax**. In this case if the signal is resumed, **CreateTable** simply returns NIL.

Error: ERROR [code: ErrorCode];

ErrorCode: TYPE = {noSuchPeriodicNotifier, other};

ReturnToNotifier: ERROR [string: XString.Reader];

Sometimes a client is deep in the call stack of some Notifier-invoked operation from which it wishes to unwind. The ERROR **ReturnToNotifier** can be raised and will be caught at the top level of the Notifier process. Clients can catch this error, post a message with **string** in it, and let the error propagate up:

55.2.15 Miscellaneous Items

GetPlace: PROCEDURE [window: Window.Handle] RETURNS [Window.Place];

GetPlace returns the **window** relative coordinate of the last user action that was matched.. **GetPlace** should only be invoked while in the notifier process.

actionToWindow: PACKED ARRAY KeyName OF BOOLEAN;

actionToWindow determines if a user action should be sent to the window containing the cursor (TRUE) or to the window containing the current input focus (FALSE). This array is global to the entire environment. It is initialized to have all actions go to the input focus, except those associated with the **Adjust**, **Menu**, and **Point** mouse buttons and the STOP key.

caretRate: Process.Ticks;

clickTimeout: System.Pulses;

clickTimeout and **caretRate** are values that are set by user profile. Clients who implement blinking carets may use **caretRate** to determing the rate of caret blinking. Clients who implement click timeout without using the timing facilities in tables may use **clickTimeout** to determine the maximum time allowed between two clicks of a multi-click.

FlushUserInput: PROCEDURE;

FlushUserInput empties the queue of pending user actions (type-ahead and button-ahead).

GetNotifierProcess: PROCEDURE RETURNS [PROCESS];

GetNotifierProcess returns the current notifier process. It is defined in TIPXX.mesa.

```
timeToFirstRepeat: Process.Milliseconds;
```

```
timeBetweenRepeats: Process.Milliseconds;
```

timeToFirstRepeat and **timeBetweenRepeats** are for clients who wish to implement repeating keys. A key is a repeating key, if while the key is held down continuously, the same action is performed each time an interval elapses. The value for the first interval is the **timeToFirstRepeat**. The value for successive intervals is **timeBetweenRepeats**.

55.2.16 "Look-Ahead"

TIP allows a client to "look ahead" at the next results that **TIP** will deliver, and choose to process the results or not. This allows the client to implement possible performance gains. For example, say the client's **NotifyProc** is called with a string result (the user typed some characters) and suppose there is some significant "setup" processing the client must do to get to the point where the characters can actually be inserted and displayed. If the next thing in the input queue is more characters, the client can save that setup processing time by looking at what's next in the input queue *before* returning from the **NotifyProc** and continuing to process characters until the next item in the queue is not a string result.

```
GetResults: PROCEDURE [window: Window.Handle, resultsWanted: ResultsWanted ← NIL]
  RETURNS [results:Results];
```

```
ResultsWanted: TYPE = PROCEDURE [
  window: Window.Handle, table: Table ← NIL, results: Results]
  RETURNS [wanted: BOOLEAN];
```

GetResults calls **resultsWanted** and if **resultsWanted** returns TRUE, then **GetResults** returns the **results** that would next go to **window**'s **NotifyProc**, and removes from the input queue the event that produced **results**. If **resultsWanted** returns FALSE, the event is left in the input queue and processed "normally." When **resultsWanted** is called, **results** is the next results and **window** is the window that those results would go to. If **table** is not NIL, then the **results** would go to the **table**'s **NotifyProc**. These items are defined in TIPX.mesa.

55.3 Usage/Examples

55.3.1 Periodic Notification

The following example shows the use of a periodic notifier for updating a display of the volume page count. The page count will be updated every 20 seconds, provided that the Notifier is not otherwise occupied.

```
window: Window.Handle ←...;
updateCount: Atom.ATOM ←Atom.MakeAtom["UpdateCount"L];
results: ResultsObject ←[next: NIL, body: atom[a: updateCount]];
pageNotifier: PeriodicNotify ←
  CreatePeriodicNotifty[window: window, results: @results, milliSeconds: 20000];

MyNotifyProc: NotifyProc = {
```

```
input: Results;
FOR input ← results, results.next DO
   WITH z: input SELECT FROM
      atom = >
         IF z.a = updateCount THEN {
            -- code to update page count on screen;}
         ELSE {
            -- code to handle other atoms};
      ENDCASE;
   ENDLOOP};
```

55.3.2 Syntax of TIP tables

Following is the BNF description for the syntax of tables. Non-terminals are boldface, terminals are non-bold Titan (e.g., FastMouse). The characters "|", ".", ";", ",", " = >", "{", and "}" in the BNF below are terminal symbols. The semantics are described in the next section.

TIPTable	:: = **Options TriggerStmt** . *Note: tables terminate with a period.*
Options	:: = **empty** | OPTIONS **OptionList** ;
OptionList	:: = **Option** | **Option** , **OptionList**
Option	:: = **SmallOrFast** | **FastOrSlowMouse**
SmallOrFast	:: = Small | Fast
FastOrSlowMouse	:: = FastMouse | SlowMouse
Expression	:: = AND **TriggerChoice** | WHILE **EnableChoice** | = > **Statement**
Statement	:: = **TriggerStmt** | **EnableStmt** | **Results**
TriggerStmt	:: = SELECT TRIGGER FROM **TriggerChoiceSeries**
EnableStmt	:: = SELECT ENABLE FROM **EnableChoiceSeries**
TriggerChoiceSeries	:: = **TriggerChoice** ; **TriggerChoiceSeries** | **TriggerChoice** ENDCASE **FinalChoice** | ENDCASE **FinalChoice**
EnableChoiceSeries	:: = **EnableChoice** ; **EnableChoiceSeries** | **EnableChoice** ENDCASE **FinalChoice** | ENDCASE **FinalChoice**
TriggerChoice	:: = **TriggerTerm Expression**
EnableChoice	:: = **EnableTerm Expression**
FinalChoice	:: = **empty** | = > **Statement**
TriggerTerm	:: = (**Key** | MOUSE | ENTER | EXIT) **TimeOut**
EnableTerm	:: = **KeyEnableList** | **PredicateIdent**
TimeOut	:: = **empty** | BEFORE **Number** | AFTER **Number**
KeyEnableList	:: = **Key** | **Key** | **KeyEnableList** *Note: the | between Key and KeyEnableList is a terminal and must be entered.*

Key	:: = KeyIdent Up | KeyIdent Down
Results	:: = ResultItem | ResultItem , Results | ResultItem Expression | { ResultItems }
ResultItems	:: = ResultItem | ResultItem ResultItems
ResultItem	:: = COORDS | BUFFEREDCHAR | CHAR | KEY | TIME | Number| String | ResultIdent
String	:: = "any sequence of characters not containing a ""
ResultIdent	:: = Ident
KeyIdent	:: = Ident
PredicateIdent	:: = Ident

55.3.3 Semantics of Tables

The whole match process can be viewed as a SELECT statement, that is continuously reading key transitions, mouse movements, or key states from the input queue. A trigger statement has the effect of looking at the next action recorded in the input queue and branching to the appropriate choice. An enable statement implies selection between the different choices according to the current state of the keyboard or the mouse keys. Trigger terms may appear in sequence, separated by AND. They might be mixed with enable terms, which in turn are characterized by the keyword WHILE. A timeout following a trigger indicates a timing condition that has to hold between this trigger and its predecessor. The number associated with the timeout expresses a time interval in milliseconds. Events starting with the same sequence of trigger and/or enable terms are expressed as nested statements. Result items may be identifiers, numbers, strings, or the keywords COORDS, BUFFEREDCHAR, CHAR, KEY, or TIME. The results of a successfully parsed event are passed to the client. Numbers are passed as LONG INTEGERs, and strings as XString.ReaderBodys. BUFFEREDCHAR and CHAR come as XString.ReaderBodys containing the character interpretation of the key involved with the event as defined by the CharTranslator. COORDS results in a Window.Place containing the mouse coordinates of the event.

Option	:: = SmallOrFast | FastOrSlowMouse
SmallOrFast	:: = Small | Fast TIP can produce its internal table in two formats: one designed for compactness (default) and one for speedy execution. This option indicates which format should be used.
FastOrSlowMouse	:: = FastMouse | SlowMouse
FastMouse	The TriggerTerm MOUSE means *all* mouse movement.
SlowMouse	The TriggerTerm MOUSE means only the most recent mouse motion (default).
Expression	:: = AND TriggerChoice | WHILE EnableChoice | = > Statement
AND TriggerChoice	match if and only if TriggerChoice is the next input event *after* the preceding choice. For example, A Down AND B Down

	means A goes down and then B goes down (with no intervening actions like A Up or mouse motion).
WHILE **EnableChoice**	match if **EnableChoice** is also true at this point. For example, A Down WHILE B Down matches if A goes down while B is down.
= > **Statement**	continue processing at **Statement** (it is used for results and common prefixes).
Statement	:: = **TriggerStmt** | **EnableStmt** | **Results**
TriggerStmt	:: = SELECT TRIGGER FROM **TriggerChoiceSeries**
EnableStmt	:: = SELECT ENABLE FROM **EnableChoiceSeries**
EnableStmt	matches if any of the **EnableChoiceSeries** *have already happened.*
TriggerStmt	matches if any of the **TriggerChoiceSeries** *have just happened.*
TriggerTerm	:: = (**Key** | MOUSE | ENTER | EXIT) **TimeOut**
Key	matches if the appropriate key transition occurs.
MOUSE	matches if there is mouse motion (useful for tracking the mouse).
ENTER	matches if the mouse enters the window.
EXIT	matches if the mouse leaves the window.
TimeOut	:: = **empty** | BEFORE **Number** | AFTER **Number**
BEFORE **Number**	matches if the associated **TriggerTerm** happens within a number ofmilliseconds of the preceding (matched) user action. For example, A Down AND B Down BEFORE 200 would match if A went down and then B went down within 1/5 second (and there were no intervening actions).
AFTER **Number**	matches if the associated **TriggerTerm** happens a number of milliseconds or more after the preceding user action. For example, A Down AND B Down AFTER 200 would match if A went down and then B went down more than 1/5 second later (and there were no intervening actions).
EnableTerm	:: = **KeyEnableList** | **PredicateIdent**
KeyEnableList	is true if any of the **Keys** are true.
Key	:: = **KeyIdent** UP | **KeyIdent** DOWN
Key	is true if the appropriate transition has happened (if this is part of a trigger term, is the current user action; if this is an enable term, it has already happened).
KeyIdent	identifies the keyboard key. The identifiers should be one of: A ... Z, One, Two, Three, ... Zero, Adjust, AGAIN, OpenQuote,

DEFAULTS, BackSlash, BS, SUBSCRIPT, SMALLER, Comma, KEYBOARD, TAB, PROPS, COPY, Minus, DELETE, MARGINS, Equal, EXPAND, FIND, HELP, ITALICS, UNDERLINE, Keyset1, Keyset2, Keyset3, Keyset4, Keyset5, LeftBracket, LeftShift, LOCK, MouseMiddle, CENTER, MOVE, NEXT, SAME, Period, Point, CloseQuote, SUPERSCRIPT, RETURN, RightBracket, RightShift, BOLD, SemiColon, Slash, Space, OPEN, PARATAB, UNDO, STOP, A8, A9, A10, A11, A12, L1, L4, L7, L10, Key47, R3, R4, R9, R10

Note: There are no names for shifted characters like left or right paren. Instead, specify one or both shift keys plus the unshifted key name. For example, Nine Down WHILE LeftShift Down instead of LeftParen Down.

See **LevelIVKeys** and the *Pilot Programmer's Manual* for (610E00160) more information on the keyboard.

ResultItem	:: = COORDS | BUFFEREDCHAR | CHAR | KEY | TIME | **String | Number | ResultIdent**
String	:: = **"any sequence of characters not containing a ""**
ResultIdent	:: = **Ident**
COORDS	return a coord **ResultElement** with the coords of the last user action.
BUFFEREDCHAR	return a string **ResultElement** containing the character representations of the last user actions that were also buffered characters.
CHAR	same as BUFFEREDCHAR
KEY	return a key **ResultElement** with the current state of the key (not recommended in normal usage. Usually a more complex TIP table is indicated if you are using this result).
TIME	return a time **ResultElement** with the time of the last (matched) user action.
String	return a string **ResultElement.**
Number	return an integer **ResultElement.**
ResultIdent	return an atom **ResultElement.**
PredicateIdent	:: = **Ident**

PredicateIdent a predicate is an atom which can have a procedure associated with it (defined in SpecialTIPX.mesa). This procedure returns a **BOOLEAN** and it is called by TIP when a predicate atom is encountered in a TIP table. Several useful predicates are provided by ViewPoint, to distinguish physical keyboards: "aLevel4", "eLevel4", "jLevel4", "aLevel5", "eLevel5", "jLevel5" (where a = American, e = European, j = Japanese, Level4 = 8010, Level5 = 6085). For example, adding this to a TIP table:

```
    TAB Down = > SELECT ENABLE FROM
        eLevelV = > SomethingSpecial
        ENDCASE;
```

would result in "SomethingSpecial" being passed in the results list if and only if the TAB key went down on a European 6085 keyboard. See NormalKeyboard.TIP in Appendix A for more examples.

55.3.4 Example Table

```
SELECT TRIGGER FROM
  Point Down => 
    SELECT TRIGGER FROM
      Point Up BEFORE 200 AND Point Down BEFORE 200 =>
        SELECT ENABLE FROM
          LeftShift Down => COORDS, ShiftedDoubleClick
          ENDCASE      => COORDS, NormalDoubleClick;
  Adjust Down BEFORE 300 => PointAndAdjust;
  ENDCASE => COORDS, SimpleClick;
  ...
```

This table produces the result element (atom) NormalDoubleClick along with the mouse coordinates if the left mouse button goes down, remains there not longer than 200 ms, and goes down again before another 200-ms elapse. The result is ShiftedDoubleClick if the same actions occur but also the left shift key is down. If the right mouse button also goes down less than 300 ms after the initial Point Down and the left mouse button also goes down, PointAndAdjust results. Finally, the table specifies the result SimpleClick (with coordinates) if Point goes down but none of the above-described succeeding actions occurs.

55.3.5 Simple TIP Client Example

This example shows a simple **TIP** client. The window acts in the following manner. If the left (Point) mouse button is depressed the window becomes the input focus and the cursor changes to its special shape. As long as the window is the input focus and the cursor is in the window, it remains the special shape but returns to the original shape when the mouse leaves the window. The place in the window where the depressed left (Point) mouse button is released is the place where text is displayed.

The procedure **InitAtoms** is part of the initialization code and creates the four atoms that the notify procedure understands. It is put in a separate procedure so the string literals will not be allocated in the global frame. The procedure **InitWindow** initializes the window by attaching the context data and setting the table and notify procedure.

This example uses the system's table from **TIPStar**. The fragments of the **NormalMouse.tip** portion of **TIPStar**'s normal table that are used to generate the atom results in this example are

```
Point Down => SELECT ENABLE FROM
  [SHIFT] => TIME, COORDS, Shift, PointDown;
  ENDCASE => TIME, COORDS, PointDown;

Point Up => SELECT ENABLE FROM
  [SHIFT] => TIME, COORDS, Shift, PointUp;
  ENDCASE => TIME, COORDS, PointUp;

ENTER => Enter;
EXIT => Exit;
```

The notify procedure **TIPMe** looks at the results and understands four atoms and string input. For the atom **pointDown**, if the window is not already the input focus, it sets the window as the input focus and sets the cursor to its special shape. For the atom **pointUp**, it saves the place the event occurred as the location to display text. The **enter** atom just fiddles with the cursor if the window is the input focus. The **exit** atom restores the cursor. If the window is the input focus and the user types into the window, a string result is sent to the notify procedure containing the characters typed. The **NormalKeyboard.tip** portion of **TIPStar**'s normal table contains the BUFFEREDCHAR results for the keyboard–keys–going down events.

```
Handle: TYPE = LONG POINTER TO Object;
Object: TYPE = ...;
contextType: Context.Type = Context.UniqueType[];
pointDown, pointUp, enter, exit: Atom.ATOM;

InitAtoms: PROCEDURE = {
  pointDown ← Atom.MakeAtom["PointDown"L];
  pointUp ← Atom.MakeAtom["PointUp"L];
  enter ← Atom.MakeAtom["Enter"L];
  exit ← Atom.MakeAtom["Exit"L]};

InitWindow: PROCEDURE [window: Window.Handle] = {
  h: Handle = zone.NEW[Object ← []];
  Context.Create[type: contextType, data: h, proc: DestroyContext, window: window];
  TIP.SetTableAndNotifyProc[
    window: window, table: TIPStar.NormalTable[], notify: TIPMe]};

TIPMe: TIP.NotifyProc = {
  h: Handle = Context.Find[type: contextType, window: window];
  place: Window.Place;
  FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
    WITH z: input SELECT FROM
      coords = > place ← z.place;
      atom = > SELECT z.a FROM
        pointDown = >
          IF ~h.hasInputFocus THEN {
            TIP.SetInputFocus[
              window: window, takesInput: TRUE,
              newInputFocus: MyLosingFocusProc, clientData: h];
            SaveCursorAndSetMine[h]}
        pointUp = > h.textPlace ← place;
        enter = > IF h.hasInputFocus THEN SaveCursorAndSetMine[h];
        exit = > RestoreCursor[h];
        ENDCASE;
      string = >
        h.textPlace ← DisplayTextAtPlace[h: h, reader: @z.rb, place: h.textPlace];
      ENDCASE;
    ENDLOOP};

MyLosingFocusProc: TIP.LosingFocusProc = {
  OPEN h: NARROW[clientData, Handle];
  h.hasInputFocus ← FALSE};
```

55.3.6 Modifying an Existing TIP Client

This example shows how an existing TIP client may be modified. Assuming the existence of a TextWindow package similiar to that in Tajo, this example builds a TTY-like window on top of it. It modifies the text window's behavior in two ways. First, it changes the table that the text window uses by linking its own table on the front of the normal table that the text window package uses. It also has its own notify procedure that just looks for the STOP key going down but passes all other notifications to the text window's notify procedure that it saves.

This example writes its own table. The table maps shift backspace to the character Control-W, backspace to the character Control-H, the DELETE key to the DEL character and the TAB key to the ESCAPE character. This table handles only a few of the functions and is linked onto TIPStar's normal table to provide the bulk of the funtion.

```
-- File: TTY.tip

[DEF SHIFT, (LeftShift Down | RightShift Down | Key47 Down | A12 Down)]

SELECT TRIGGER FROM
   BS Down = > SELECT ENABLE FROM
      [SHIFT] = > "\027";
      ENDCASE = > "\010";

   DELETE Down = > "\177";

   TAB Down = > "\033";

   ENDCASE.

Handle; TYPE = LONG POINTER TO Object;
Object: TYPE = ...;
contextType: Context.Type = Context.UniqueType[];
stop: Atom.ATOM;

Init: PROCEDURE = {
   rb: XString.ReaderBody ← XString.FromSTRING["TTY.tip"L];
   stop← Atom.MakeAtom["Stop"L];
   myTable ← TIP.CreateTable[file: @rb];
   [] ← TIP.SetTableLink[from: myTable, to: TIPStar.NormalTable[]};

Create: PROCEDURE [window: Window.Handle,...] = {
   h: Handle = zone.NEW[Object ← []];
   TextWindow.Create[window,...];
   h.oldNotify ← TIP.SetNotifyProc[window: window, notify: TIPMe]};

TIPMe: TIP.NotifyProc = {
   h: Handle = Context.Find[type: contextType, window: window];
   WITH z: results SELECT FROM
      atom = > SELECT z.a FROM
         stop = > {
            TIP.FlushUserInput[];
            SendHaltNotification[h];
            RETURN};
```

```
        ENDCASE;
    h.oldNotify[window, results]}; -- normally pass results to text window's notify
```

55.3.7 Macro Package

The macro package used in **TIP** is based on the general-purpose macrogenerator described by Strachey in *Computer Journal* (October 1965). The following summary is based on that article; see the article itself for more details.

A macro call consists of a macro name and a list of actual parameters, each separated by a comma. The name is preceded by a left square bracket ([), and the last parameter is followed by a right square bracket. A macro is defined by the special macro DEF, which takes two arguments: the name of the macro to be defined and the defining string. The defining string may contain the special symbols ~1, ~2, etc., which stand for the first, second, etc., formal parameters. Enclosing any string in parentheses prevents evaluation of any macro calls inside; in place of evaluation, one layer of string quotes is removed. It is usual to enclose the defining string of a macro definition in string quotes to prevent any macro calls or uses of formal parameters from being effective during the process of definition.

Here are some sample macros and an example:

```
-- macro definitions
[DEF,LSHIFT,(LeftShift Down)]
[DEF,RSHIFT,(RightShift Down)]
[DEF,EitherShift,(
  [LSHIFT] = > ~1;
  [RSHIFT] = > ~1)]

-- trigger cases
SELECT TRIGGER FROM
BS Down = > SELECT ENABLE FROM
  [EitherShift,{BackWord}];
  ENDCASE = > {BackSpace};
-- more cases ...
ENDCASE...
```

The above example expands to:

```
BS Down = > SELECT ENABLE FROM
  LeftShift Down = > BackWord;
  RightShift Down = > BackWord;
  ENDCASE = > {BackSpace};
```

55.4 Index of Interface Items

56

TIPStar

56.1 Overview

The **TIP** facility provides a mechanism that links a list of **TIP.Tables**. These **TIP.Tables** contain productions that translate user actions into terms a client is prepared to deal with. **TIPStar** creates a structure for the list of ViewPoint TIP tables to be built on. This structure divides all possible input actions into logical groups (mouse actions, special keys like UNDO and STOP, utility keys like MOVE and COPY, etc.) and provides a means for accessing these groups of tables.

56.2 Interface Items

56.2.1 The TIPStar Structure

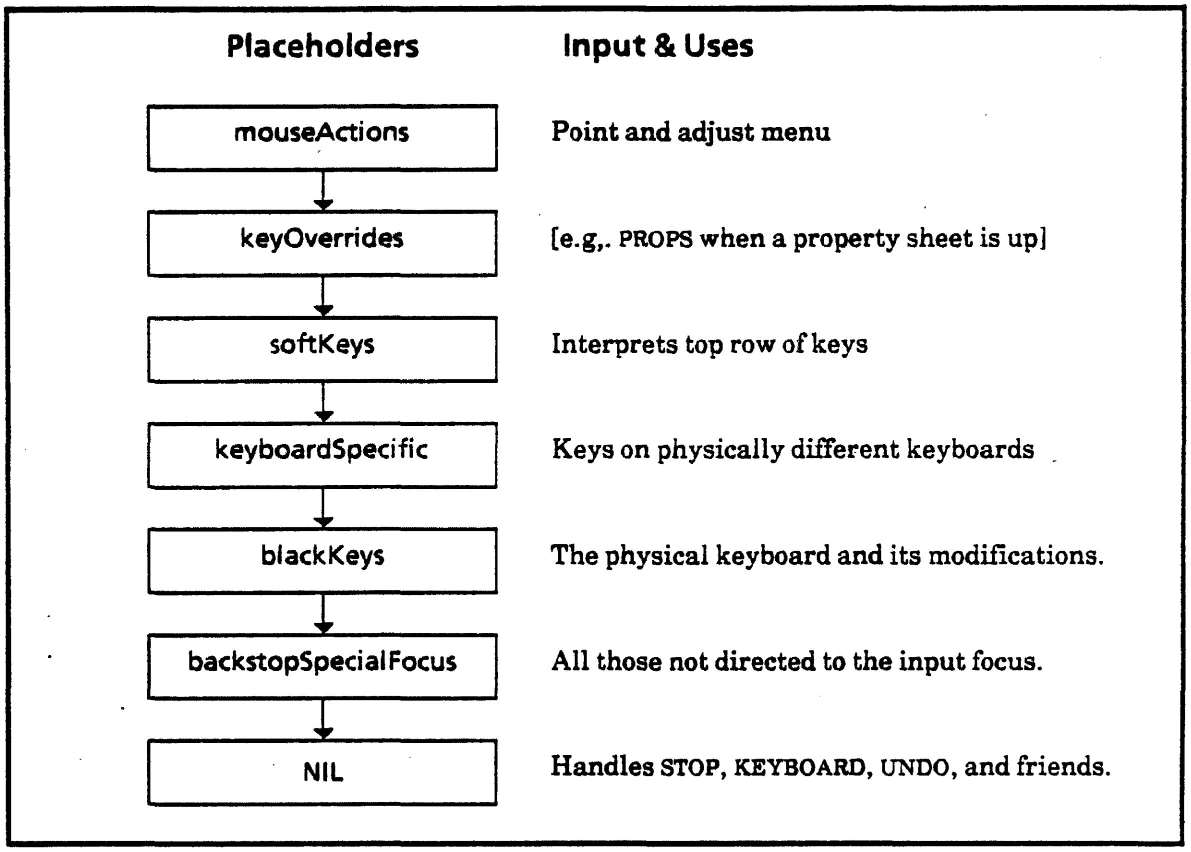
The basis for the **TIPStar** structure is the placeholder.

Placeholder: TYPE = {mouseActions, keyOverrides, softKeys, keyboardSpecific, blackKeys, sideKeys, backstopSpecialFocus};

A placeholder table is created for each of the enumerateds in **Placeholder**. Placeholder tables are *empty* TIP tables linked to form a list. This list divides all possible input actions into logical groupings as discussed in the Overview above. It defines a series of segments for the list of **TIP.Tables** to be built upon. Segments (mini-stacks) are delineated by the placeholder tables. This initial list of **TIP.Tables** then, contains only empty tables. Note: Placeholder tables are always empty. They are, as their name implies, placeholders--each providing a position in the list of tables for adding or removing real tables of a particular kind (those relating to mouse actions, those mentioning the soft keys,and so forth.). See Examples in the next section.

Fine point: A set of normal tables that contain all the basic key productions is installed at boot time. See the System TIP Tables Appendix for listings of those tables and a view of the TIP table list at the completion of booting. These normal tables are referred to as *generic* in the description of **TIPStar.GetTable** to prevent confusion with the procedure **TIPStar.NormalTable**.

The list of ViewPoint placeholder tables is initialized as in Figure 56.1 (the arrows represent the links of the list).



Placeholders	Input & Uses
mouseActions	Point and adjust menu
keyOverrides	[e.g,. PROPS when a property sheet is up]
softKeys	Interprets top row of keys
keyboardSpecific	Keys on physically different keyboards
blackKeys	The physical keyboard and its modifications.
backstopSpecialFocus	All those not directed to the input focus.
NIL	Handles STOP, KEYBOARD, UNDO, and friends.

Figure 56.1 ViewPoint Placeholder Tables

56.2.2 Installing and Removing Tables

A client may alter the table arrangement by *pushing* or *storing* a TIP table into any point on the tree, or by *popping* back to a previous table.

PushTable: PROCEDURE [Placeholder, TIP.Table] ;

PushTable leaves old tables in the watershed, but places the new table (or chain of tables) directly after the specified placeholder. This places the new table in front of any others within that segment. Thus if the new table mentions the same key actions as the old table, the old one is effectively ignored until the new one is popped. If the new table only mentions a few key actions, however, previously pushed tables will be used for the others.

For an example of **PushTable** and the resulting **TIP.Table** list, see §56.3.1.

PopTable: PROCEDURE [Placeholder, TIP.Table] ;

PopTable takes the single TIP table to be popped. It is not required that the table to be popped be at the top of the placeholder's list. A strict stack discipline is relaxed.

StoreTable: PROCEDURE [Placeholder, TIP.Table] RETURNS [TIP.Table] ;

StoreTable replaces the table (or chain of tables) with the client's table (or chain of tables) and returns the previous table. The client can restore the old value later, if it wishes. (In using **StoreTable**, and especially in remembering or restoring the old value, the client probably needs to be cognizant of the other clients that may manipulate the same placeholder.) (See examples in §56.3.3.)

56.2.3 Retrieving Pointers to Installed Tables

NormalTable: PROCEDURE RETURNS [TIP.Table];

NormalTable returns the table at the head of the list (**mouseActions** placeholder). This is the appropriate table to use for a normal **TIP.SetTableAndNotifyProc**.

GetTable: PROCEDURE [Placeholder] RETURNS [TIP.Table];

GetTable returns the generic table at the specified placeholder, if one exists. (See the fine point in §56.2.1.)

56.2.4 Mouse Modes

Mode: TYPE = {normal, copy, move, sameAs};

The **TIPStar.Modes** refer to the various modes attributable to mouse actions. These modes can be programmatically checked and changed by using the **GetMode** and **SetMode** procedures outlined below.

GetMode:PROCEDURE RETURNS [mode: Mode];

GetMode returns the current mode.

SetMode:PROCEDURE [mode: Mode] RETURNS [old: Mode];

Calling **SetMode** causes the appropriate **TIP.Table** to be stored in the **TIPStar** chain.

For example, when the COPY key goes down, the call to **TIPStar.SetMode[copy]** causes **NormalMouse.TIP** to be relaced by **CopyModeMouse.TIP**. Clients receiving mouse notifications receive **CopyModeDown** instead of **PointDown**. If the world is in **move** mode (causing **MoveModeMouse.TIP** to be stored) the client receives the **MoveModeDown** when mouse point is pressed. See the TIP Table appendix for information on the other productions in the four mouse tables (**NormalMouse.TIP**, **CopyModeMouse.TIP**, **MoveModeMouse.TIP** and **SameAsModeMouse.TIP**).

56.3 Usage/Examples

56.3.1 When PushTable Is Called

```
InitializeMyTIPTables: PROCEDURE =
BEGIN
rbClientAMouse: XString.ReaderBody ←XString.FromSTRING["ClientAMouse.TIP"L];
 tipClientAMouse: TIP.Table ←TIP.CreateTable[file: @rbClientAMouse];
 -- install my tip table (tie it to my notify proc)
[]←TIP.SetNotifyProcForTable[ tipClientAMouse, ClientAMouseNotifyProc];
PushTable[mouseActions, tipClientAMouse];

rbClientAKeys: XString.ReaderBody ←XString.FromSTRING["ClientAKeys.TIP"L];
 tipClientAKeys: TIP.Table ←TIP.CreateTable[file: @rbClientAKeys];
 -- install my tip table (tie it to my notify proc)
[]←TIP.SetNotifyProcForTable[ tipClientAKeys, ClientAKeysNotifyProc];
PushTable[sideKeys, tipClientAKeys];
END; -- InitializeMyTIPTables
```

Assume initially that the list appears as in Figure 56.1. If Client A then pushes two tables onto that list, as in the code above, the new links result in the list shown in Figure 56.2..

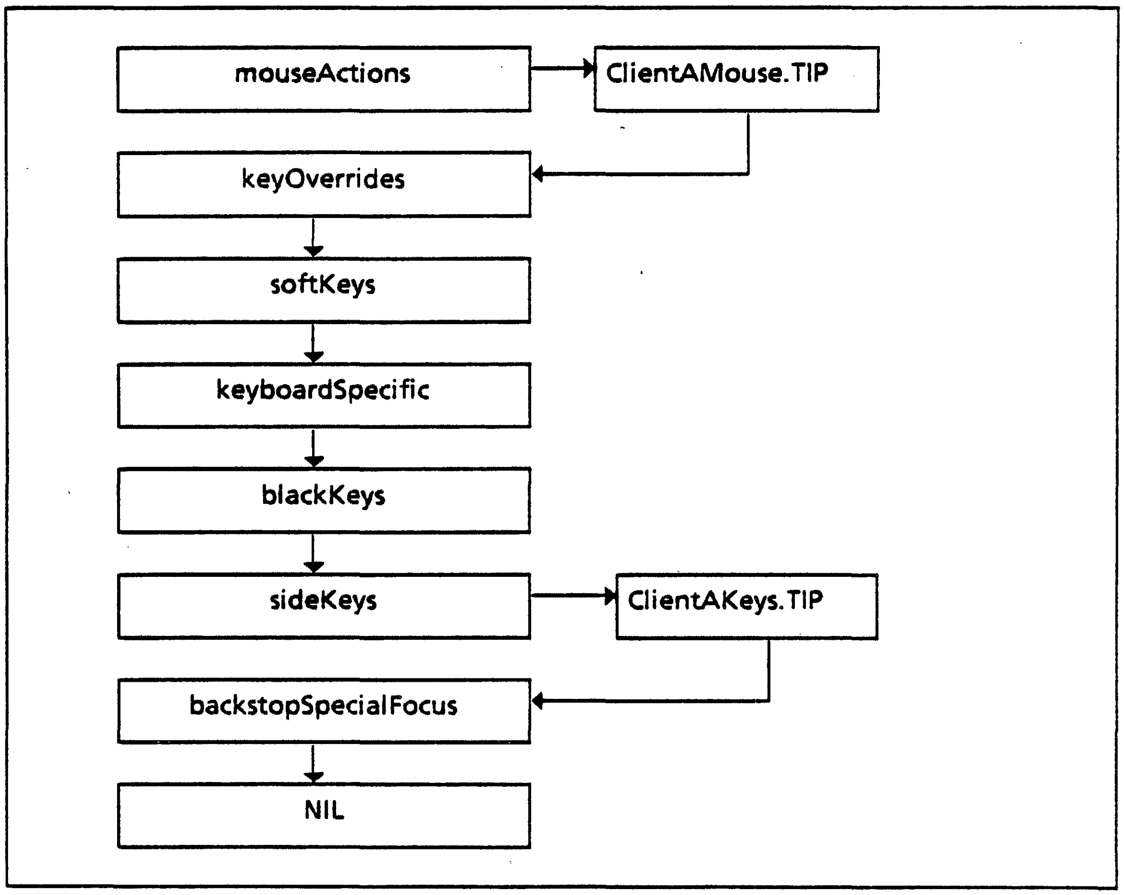


Figure 56.2 When PushTable Is Called

If client B then pushes another table to the **mouseActions** placeholder

```
PushTable[mouseActions, tipClientBMouse];
```

the resulting list appears as in Figure 56.3.

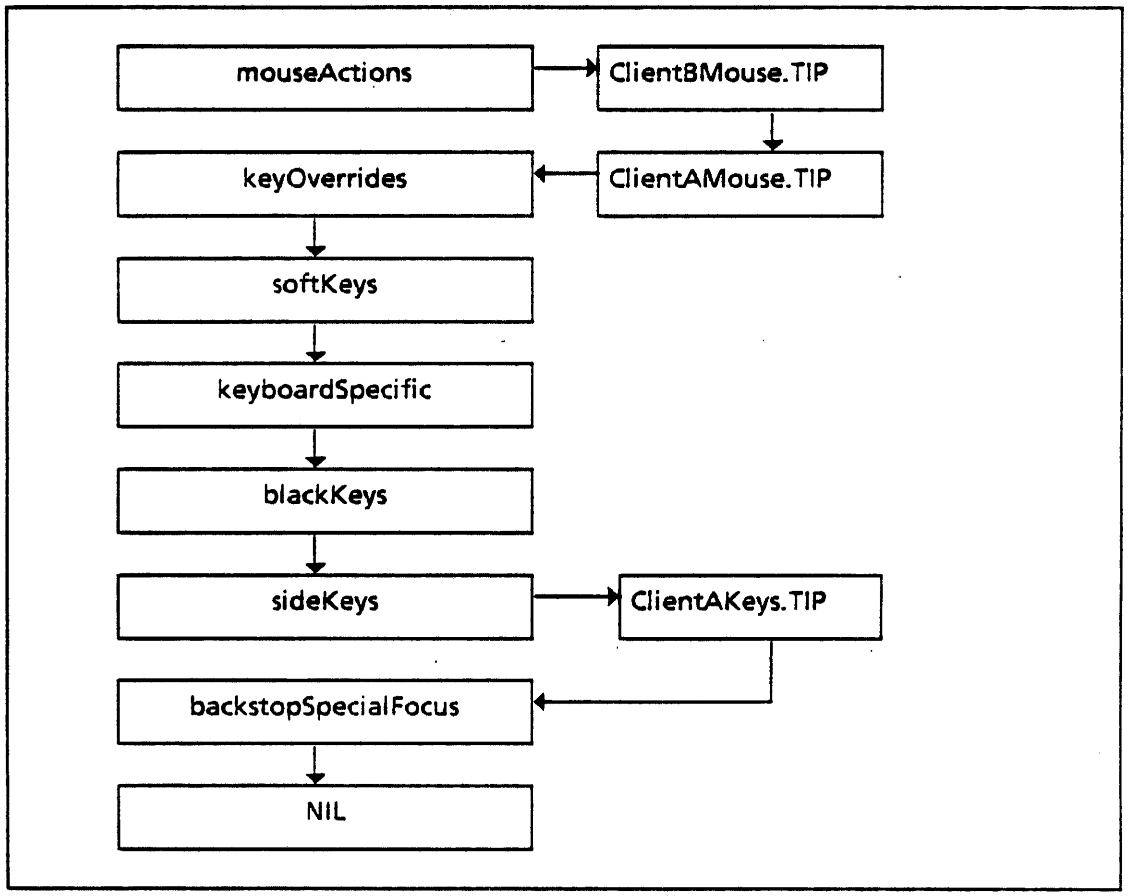


Figure 56.3 Pushing Another Table

56.3.2 When StoreTable Is Called

```
rbClientCMouse: XString.ReaderBody ← XString.FromSTRING["ClientCMouse.TIP"L];
tipClientCMouse: TIP.Table ← TIP.CreateTable[file: @rbClientCMouse];
-- install my tip table (tie it to my notify proc)
[]←TIP.SetNotifyProcForTable[ tipClientCMouse, ClientCMouseNotifyProc];
savedTable ← StoreTable[mouseActions, tipClientCMouse];
```

Assume initially that the list appears as in Figure 56.3. If client C then calls **StoreTable** with another table directed at the **mouseActions** placeholder, the resulting list appears as in Figure 56.4.

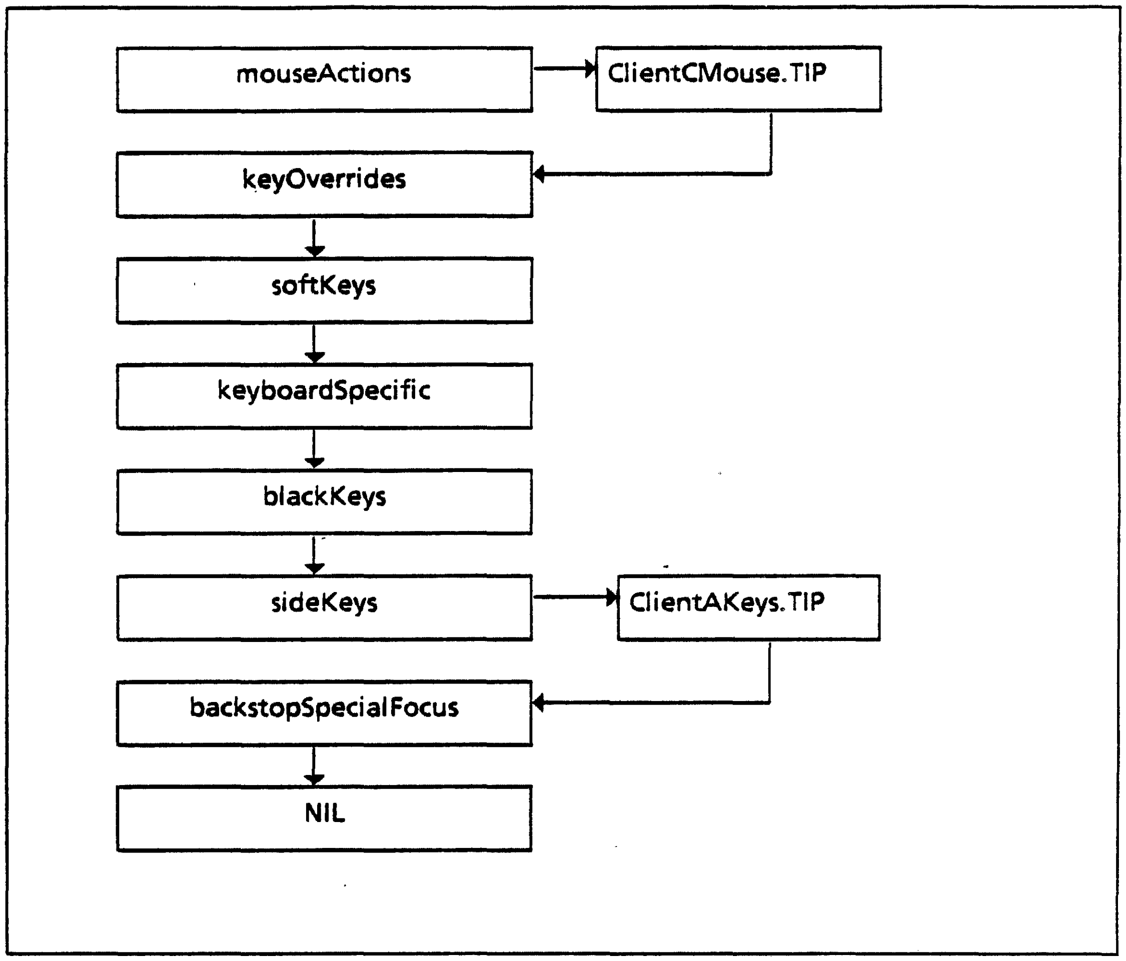


Figure 56.4 Pushing Another Table

Client C now has the handle to the segment removed from **mouseActions** when the **StoreTable** was done (**savedTable**, see Figure 56.5). This table (or in this case, chain of tables) should be replaced when the client is through with its own mouse tip (**tipClientCMouse**) by a call to:

StoreTable[mouseActions, savedTable]; or
PopTable[mouseActions, tipClientCMouse];
PushTable[mouseActions, savedTable];

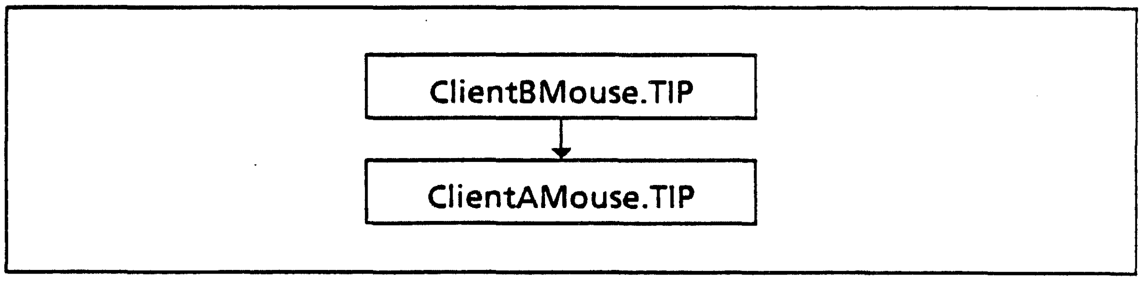


Figure 56.5 Saved Table

56.3.3 When PopTable Is Called

Assume initially that the list appears as in Figure 56.3. If client A then pops its table at the **mouseActions** placeholder, the resulting list appears as in Figure 56.6. **Note**: It is not necessary for the table being popped to be at the top of the stack (where the top of a stack is here defined to be the position immediately following any placeholder table--thus there are several stacks within the watershed list of tables).

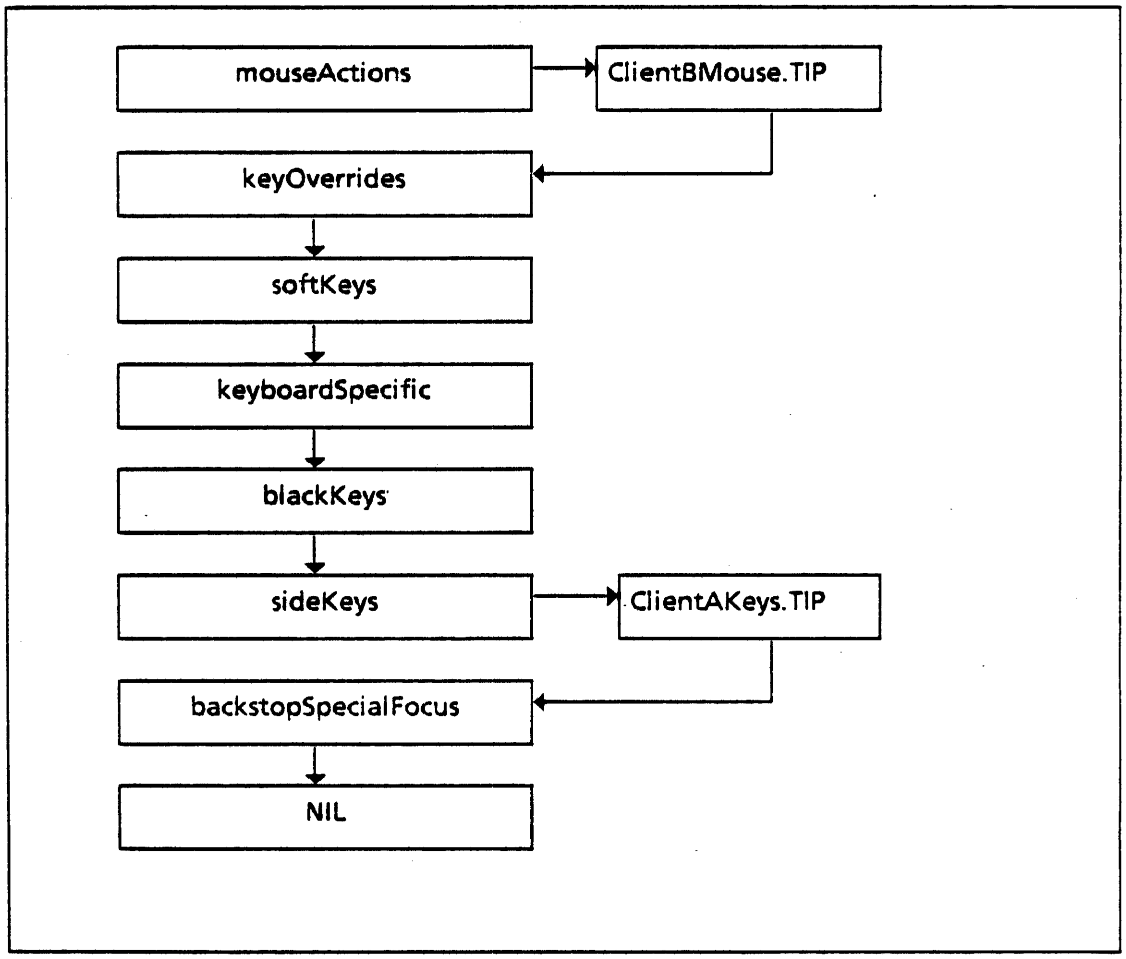


Figure 56.6 Pop Table

56.4 Index of Interface Items

57

Undo

57.1 Overview

The **Undo** interface provides a set of procedures that allow applications to add **Undo** opportunities to the current **Undo** stack. An implementation of **Undo** can then call applications when the UNDO key is depressed.

57.2 Interface Items

57.2.1 Application's Procedures

```
Opportunity: Undo.Proc;

Proc: TYPE = PROCEDURE [
   undoProc: PROCEDURE [LONG POINTER],
   destroyProc: PROCEDURE [LONG POINTER],
   data: LONG POINTER,
   size: CARDINAL ← 0 ];
```

The **Opportunity** procedure is called by an application when it does something that can be undone. The client's **undoProc** is called to perform an undo. The **destroyProc** is called when the undo opportunity no longer exists. The client can destroy any data at that time. The **undoProc** or the **destroyProc** will always be called. The client's context for the undoing is passed in via the **data** item: a non-zero **size** indicates that the **Undo** implementation should copy the words from **data↑** through **(data + size-1)↑** into its zone. If **size** = 0, the caller's long pointer is simply remembered.

```
Roadblock: PROCEDURE [XString.Reader];
```

The **Roadblock** procedure is called by an application when it does something that cannot be undone. The immutable string passed in is a message that the **Undo** implementation can issue if the user attempts to undo past this point. The string is copied.

```
DoAnUndo: PROCEDURE;
```

The **DoAnUndo** procedure is called when an undo action should be forced. This is typically when the keyboard modules notice that UNDO has been pressed.

```
DoAnUnundo: PROCEDURE;
```

The **DoAnUnundo** procedure is called when an un-undo action should be forced. This is typically when the keyboard modules notice that shift-UNDO has been pressed.

```
DeleteAll: PROCEDURE;
```

The **DeleteAll** procedure is called to tell the **Undo** implementation to empty its stack of opportunities. This procedure is typically called upon logoff.

57.2.2 Implementation's Procedures

```
SetImplementation: PROCEDURE [
    Undo.Implementation] RETURNS [Implementation];

GetImplementation: PROCEDURE RETURNS [Implementation];

Implementation: TYPE = RECORD [
   opportunity: Undo.Proc,
   roadblock: PROCEDURE [XString.Reader],
   doAnUndo: PROCEDURE,
   doAnUnundo: PROCEDURE,
   deleteAll: PROCEDURE ];
```

These procedures allow an implementation to plug itself in to the **Undo** mechanism. An implementation can supply its set of procedures and can ascertain the current procedures. **SetImplementation** returns the procedures of the previous implementation.

An initial set of dummy procedures are provided. They are basically no-ops; the dummy **Opportunity** procedure immediately calls the application's **opportunity.destroyProc**.

```
Zone: PROCEDURE RETURNS [UNCOUNTED ZONE]
```

Returns the implementation's zone.

57.3 Usage/Examples

The application calls **Opportunity** with some context. The **Undo** implementation eventually calls the application either at its **undoProc** or its **destroyProc**. The former is called upon a real undo request. The latter is called when the opportunity is about to be forgotten: it allows the application to garbage-collect context. The **destroyProc** is typically called to prune the undo stack of very old elements or to prune opportunities that are trapped behind a roadblock.

At the application's **undoProc** or **destroyProc**, the argument is either (1) the original pointer passed in to **Opportunity**, if size was zero or (2) a pointer into the **Undo** implementation's zone that points to a copy of the application's data. In the latter case, the data is freed by the **Undo** implementation right after the call. Exception: if the help implementation is a no-op implementation, it can call the application's **destroyProc** from inside the **Opportunity** call. In this case, the help implementation can present the original pointer to the **destroyProc** even if size is non-zero.

57.3.1 Example

```
MyUndoDataObject: TYPE = RECORD [...];
MyUndoData: TYPE = LONG POINTER TO MyUndoDataObject;
complaint: XString.ReaderBody ← XString.FromSTRING ["Can't do that"L];

UndoProc: PROCEDURE [myUndoData: MyUndoData] = {
   -- does something appropriate, like a partial cleanup of data structures.message
   -- might post a about current state for the user.
   Undo.Zone [].FREE [@myUndoData];
   };

DestroyProc: PROCEDURE [myUndoData: MyUndoData] = {
   Undo.Zone [].FREE [@myUndoData];
   };

-- Mainline code
-- Code that cannot be undone
...
Undo.Roadblock [@complaint];
-- Code that can be undone
...
Undo.Zone[].NEW[MyUndoDataObject ← [...] ];
Undo.Opportunity [undoProc: UndoProc, destroyProc: DestroyProc, data: myData];
...
```

57.4 Index of Interface Items

58

UnitConversion

58.1 Overview

UnitConversion provides for converting numbers between various units of measure.

58.2 Interface Items

```
Units: TYPE = {inch, mm, cm, mica, point, pixel, pica, didotPoint, cicero};
```

Units defines all the units that may be converted. point is printer point. pixel is screen dot. pica = 12 points.

```
ConvertReal: PROCEDURE [n: XLReal.Number, inputUnits, outputUnits: Units]
    RETURNS [XLReal.Number];
```

ConvertReal converts n from inputUnits to outputUnits, using XLReal. May raise XLReal.Error.

```
ConvertInteger: PROCEDURE [n: LONG INTEGER, inputUnits, outputUnits: Units]
    RETURNS [LONG INTEGER];
```

ConvertInteger converts n from inputUnits to outputUnits. May raise XLReal.Error.

58.3 Usage/Examples

58.3.1 Converting Font Values

The following example implements a real-number conversion utility:

```
Unit: TYPE = MACHINE DEPENDENT {inch(0), mm(1), mica(2), point(3), space(4), cm(5), (15)};

Convert: PUBLIC PROC [n: XLReal.Number, inputUnits, outputUnits: Unit.Units]
  RETURNS [XLReal.Number] = {
  IF inputUnits = outputUnits THEN RETURN [n];
  IF inputUnits = space THEN
    RETURN
      UnitConversion.ConvertReal[
            XLReal.Multiply[n, pointPerSpace],seventySecondOfAnInch,
                ConvertUnits[outputUnits]];
    IF outputUnits = space THEN
        RETURN
            XLReal.Divide[UnitConversion.ConvertReal[n, ConvertUnits[inputUnits],
                seventySecondOfAnInch], pointPerSpace];
    RETURN
        UnitConversion.ConvertReal[
            n, ConvertUnits[inputUnits], ConvertUnits[outputUnits]]};

ConvertUnits: PROC [u: Units] RETURNS [UnitConversion.Units] = {
    IF u < mica THEN RETURN [VAL[u.ORD]];
    IF u = mica THEN RETURN [VAL[u.ORD + 1]];
    IF u = point THEN RETURN [VAL[u.ORD + 6]];
    RETURN [cm]
    };
```

58.4 Index of Interface Items

59

Window

59.1 Overview

The **Window** interface supplies facilities for managing windows on the display screen. A *window* is a rectangular region of the display screen in which a client can display information to the user. A window may overlap another window or even completely cover it. A window may extend past the edges of the physical display screen or even be completely outside it and thus not visible. Windows may be moved around horizontally and vertically, have their size changed, and have their depth changed in the stack of windows visible on the screen. **Window** shields the client from these considerations–from the client's point of view, each window is unaffected by other windows or by the edges of the display screen. **Window** automatically handles client requests to paint into window regions that are not currently visible on the screen.

The **Display** interface supplies routines for painting into windows.

59.1.1 Window Creation

Window supplies operations to allocate and free a window (a **Window.Object**). However, windows are usually not allocated directly by clients but are obtained from various other facilities, such as **StarWindowShell** or **FormWindow**. Once allocated, a window is referred to and manipulated by reference, using a **Window.Handle**.

59.1.2 Child Windows and the Window Tree

Window manipulates a *tree* of windows. A window may have *child windows*. Child windows obscure their parent; that is, they are above their parent in the apparent stack of windows visible on the screen. A child window may be entirely contained within its parent's screen area, may project beyond its parent's edges, or may even be completely outside its parent. **Window** automatically clips the display of a child window at its parent's edges. Thus a child window that is completely outside its parent is not visible on the screen at all.

Each window has an ordered list or *stack* of its child windows. Sibling windows may overlap: if they do, one that appears earlier in the stack is on top of or obscures one that appears later. The first window in the stack is the *top sibling*, and the last is the *bottom* *sibling*. Each window has a pointer to its parent, a pointer to the next sibling of its parent, and a pointer to the window's topmost child.

When a window is created, it is not in the window tree and is called a *private window*. A private window is unknown to **Window** and is not displayed on the screen. **Window** provides facilities for inserting private windows into the tree, moving them within the tree, and removing them from it. A window that is in the tree will be wholly or partially visible on the screen unless it is entirely outside its parent's area or unless its children completely cover the portion that is within its parent. Private windows may also be built into *private trees*, which can be inserted into and removed from the window tree as a unit.

Display supplies the *root window*, which is the root of the window tree and corresponds to the entire display screen. The root window typically supplies the background pattern.

Each window has its own *coordinate system*: the upper-left corner is the origin [x: 0, y: 0], with x increasing to the right and y increasing downward. A window's location is defined in terms of its parent's window coordinate system. Coordinates may be positive or negative, and thus a window can have any location relative to its parent.

59.1.3 Painting into a Window

Every window contains a client-supplied *display procedure* that will, on demand, paint all or part of the window. Note that windows can be much larger than the display screen; any paint directed to non-visible portions of a window (or outside the window entirely) is discarded. Thus, a client never needs to be concerned about what parts of its window are covered by other windows or what parts are off the screen. As a convenience to clients, requests to paint into a window that is not currently in the window tree are also ignored.

The **Display** and **SimpleTextDisplay** interfaces provide a variety of procedures for painting various things into a window, including character strings, black, white, or gray boxes, and various graphics, such as curves and lines.

The display background color, which is represented by a pixel value of zero, is commonly called *white* and a value of one is called *black*. **Note:** The display hardware also can render the picture using zero for black and one for white. *Clearing* or *erasing* an area of the screen means setting all of its pixels to zero, or white.

A display procedure usually wants to start with an erased (zero) area and logically OR the black pixels into the area. **Window** supplies an accelerator **clearingRequired** to minimize unnecessary erasures. If **clearingRequired** = TRUE, **Window** guarantees that when the display procedure is called to paint the window, all of the window's pixels that should be white indeed are white. In that situation, the window might contain any combination of its previous contents and erased areas. On the other hand, some display procedures want to set all pixel values, completely overwriting the previous contents. These windows should specify **clearingRequired** = FALSE.

Areas displayed on the screen may become incorrect or *invalid* for many reasons, such as when a window that was visible is deleted. A client can also mark an area invalid. **Window** accumulates these *invalid areas* and then, in response to a client call to **Validate** or **ValidateTree**, calls the various windows' display procedures to paint the necessary areas. ***Validate** and **ValidateTree** are the only **Window** operations that cause immediate screen* *painting*. All other operations merely enqueue work to be performed by a later **Validate** operation. Fine point: The few special cases that do not follow this rule are noted in the text.

The standard way for a client to paint into its window is to update its data structures, invalidate the portion of its window that needs to be painted, and then call a **Validate** routine. **Window** responds by calling back into the client's display procedure to do the painting.

When a window's display procedure is called, it has access to a list of the invalid areas of the window (see **EnumerateInvalidBoxes**). It may choose to paint the entire window or, alternatively, to enumerate the invalid areas and just paint those areas. In any case, **Window** clips all the display routine's paint requests to the boundaries of the invalid areas–paint directed to other areas is discarded. In special circumstances, the client may wish to paint into valid visible areas. The operation **FreeBadPhosphorList** deletes the display routine's invalid area list; for the lifetime of that invocation of that display procedure, paint requests are clipped only to the boundaries of the visible parts of the window.

If display routines are called from outside the invocation of a window's display procedure, the paint requests will be clipped to the boundaries of the visible parts of the window.

59.1.4 Bitmap-under

The window package allows clients to associate a window with a *bitmap-under*. This is a block of memory that is used to hold the pixels that are covered up by the window. It allows **Window** to move or delete such a window quickly, since it can repaint the display directly by using the contents of the bitmap-under instead of calling client display procedures. A bitmap-under is commonly used for menu windows. This is discussed in greater detail under §59.2.7.

59.1.5 Window Panes

The window package normally maintains a detailed list of invalid regions and allows arbitrary overlapping of windows without requiring the client to worry about other windows. Some clients would prefer to have greater control over their windows at the expense of more restrictions over their use. Window panes are such a mechanism. If a window is a window pane, the client must ensure that it does not overlap any of its siblings and that the parent does not paint underneath the pane. A further restriction is that only window panes may be children of panes. In return, the window package can do much less calculation to determine invalid regions. The window package does not enforce these restrictions. It is up to the client to follow them, or the screen appearance maybe inconsistent. The client must specify whether a window is a window pane when it is initialized.

59.1.6 Linked Windows

The window package allows multiple windows to linked together to make it easy for client software to support displaying the same data into multiple windows. *Linked Windows* are designed so that whatever data is painted into one window will be painted into all of the other windows. This is discussed in greater detail under §59.2.8.

59.1.7 Buffer Backed Windows

Occasionally a client may need to capture the bitmaps created by procedures in the **Display** interface rather than having them painted on the screen. This is accomplished by creating a special window handle, a *Buffer Backed Window*, which is passed to procedures in the **Display** interface. This is discussed in greater detail under §59.2.9.

59.2 Interface Items

59.2.1 Basic Data Types and Utility Operations

This section describes basic **Window** data types and utility procedures.

Handle: TYPE = LONG POINTER TO **Object**;

Object: TYPE [19];

Object is the storage that represents a window. A **Handle** is used to refer to the window. Clients should not allocate objects directly but must use operations described in §59.2.2.

rootWindow: READONLY **Handle**;

Root: PROCEDURE RETURNS [**Handle**] = INLINE {RETURN[**rootWindow**]};

rootWindow is the window that is the root of the window tree. The procedure **Root** is provided for compatibility with previous versions; new applications should use **rootWindow** instead.

MinusLandBitmapUnder: TYPE [6];

MinusLandBitmapUnder is additional storage for windows that may have bitmap-unders.

MinusLandColor: TYPE [1];

MinusLandColor is not used in the current release.

MinusLandCookieCutter: TYPE [2];

MinusLandCookieCutter is not used in the current release.

Place: TYPE = UserTerminal.**Coordinate**; -- *[x, y: INTEGER];*

Place is a position in a window. It is measured relative to the window's upper-left corner, which is defined to be at [**x**: 0, **y**: 0]. **x** increases to the right, **y** increases downward. Note that the coordinates may be negative.

Dims: TYPE = RECORD [**w**, **h**: INTEGER];

Dims is the size of a rectangular box. The rectangle is **w** pixels wide and **h** pixels high.

Box: TYPE = RECORD [**place**: **Place**, **dims**: **Dims**];

```
BoxHandle: TYPE = LONG POINTER TO Box;
```

```
nullBox: Box = [place: [0, 0], dims: [0, 0]];
```

Box describes completely a rectangular box. **place** describes the upper-left pixel of the box, and **dims** describes the size of the box. The box extends to the right and downward from **place**. As always, **place** is expressed in its containing window's coordinate system.

```
BoxesAreDisjoint: PROCEDURE [a, b: Box] RETURNS [BOOLEAN];
```

BoxesAreDisjoint returns TRUE if **a** and **b** do not intersect.

```
IntersectBoxes: PROCEDURE [b1, b2: Box] RETURNS [box: Box];
```

IntersectBoxes returns a **Box** that is the intersection of **b1** and **b2**. If their intersection is empty, this operation returns **box.dims = [0, 0]**.

```
IsPlaceInBox: PROCEDURE [place: Place, box: Box] RETURNS [BOOLEAN];
```

IsPlaceInBox returns TRUE if **place** is a pixel of **box**.

```
BitmapPlace: PROCEDURE [window: Handle, place: Place ← [0,0]] RETURNS [Place];
```

BitmapPlace returns the coordinates in the root window that correspond to **place** in **window**.

```
BitmapPlaceToWindowAndPlace: PROCEDURE [bitmapPlace: Place]
   RETURNS [window: Handle, place: Place];
```

BitmapPlaceToWindowAndPlace returns the topmost visible window and the coordinates within it that correspond to **bitmapPlace** in the root window.

59.2.2 Window Creation and Initialization

A window is created by the client allocating and initializing a **Window.Object**. Many times windows are not created directly by clients, but rather are obtained from various other facilities, such as **StarWindowShell** or **FormWindow**.

To create a window, the client allocates a **Window.Object** using **New**, initializes it using **Initialize**, and presents it to **Window** for use using **InsertIntoTree**. When the window is of no further use, it is withdrawn from **Window** using **RemoveFromTree**, and the storage is freed using **Free** or **FreeTree**.

```
New: PROCEDURE [
   under, cookie, color: BOOLEAN ← FALSE, zone: UNCOUNTED ZONE ← NIL] RETURNS [Handle];
```

New allocates a window object. If **zone** is NIL, a cache of objects is used. A client should never call **zone.NEW[Window.Object]** because the window object will not be properly initialized.

```
Initialize, InitializeWindow: PROCEDURE [
   window: Handle, display: DisplayProc, box: Box,
   parent: Handle ← rootWindow, sibling, child: Handle ← NIL,
   clearingRequired: BOOLEAN ← TRUE, windowPane: BOOLEAN ← FALSE,
   under, cookie, color: BOOLEAN ← FALSE];
```

```
DisplayProc: TYPE = PROCEDURE [window: Handle];
```

Initialize and **InitializeWindow** initialize the window object at **window↑**. This must be done before the window is inserted into the window tree. The window is initially not a part of the window tree. It may be created as an isolated window or may be linked to other private windows to form a private tree.**display** is the client procedure for repainting the window. **box** is the window's size and parent-relative location. **parent** is the window's parent. **sibling** is the sibling immediately below the window in the sibling stack of **parent** and **child** is the top child of the window. **parent**, **sibling**, and **child** may be NIL. **clearingRequired** is described in §59.1.3. **windowPane** is described in §59.1.4. **under** indicates that the window can be associated with a bitmap-under. **cookie** and **color** are not supported in the current release; clients should default this parameter for compatibility with future versions.

```
Create: PROCEDURE [
   display: DisplayProc, box: Box,
   parent: Handle ← rootWindow, sibling, child: Handle ← NIL,
   clearingRequired: BOOLEAN ← TRUE, windowPane: BOOLEAN ← FALSE,
   under, cookie, color: BOOLEAN ← FALSE, zone: UNCOUNTED ZONE ← NIL]
   RETURNS [Handle] = INLINE ...;
```

Create is an inline that follows a call to **New** with a call to **Initialize**.

```
Free: PROCEDURE [window: Handle, zone: UNCOUNTED ZONE ← NIL];
```

Free frees a window object. If **zone** is NIL, the window is returned to the cache of objects maintained by **Window**; otherwise it is freed to the zone. Any contexts associated with the window, via the **Context** interface, are not freed. **Free** may raise **Error[invalidParameters]** if the window had already been freed, if the window is still in the window tree, if the zone is NIL but was not NIL on the call to **New**, or if the zone is non–NIL but was NIL on the call to **New**.

```
FreeTree: PROCEDURE [window: Handle, zone: UNCOUNTED ZONE ← NIL];
```

FreeTree frees the window and all its children, children first, and frees all contexts on windows in the subtree whose root is **window**.. Clients should almost always call **FreeTree** rather than **Free**. **FreeTree** may raise **Error[invalidParameters]** if the windows had already been freed, if the windows are still in the window tree, if the zone is NIL but was not NIL when the windows were allocated, or if the zone is non–NIL but was NIL when the windows were allocated. **FreeTree** assume all the windows were allocated with the same zone if it is non–NIL.

59.2.3 Access to and Modification of a Window's Properties

The **Get** procedures below return properties of a window. The **Set** procedures change properties and return the previous value. These properties of a window are described in this chapter's overview.

GetDisplayProc: PROCEDURE [Handle] RETURNS [DisplayProc];

SetDisplayProc: PROCEDURE [Handle, DisplayProc] RETURNS [DisplayProc];

GetClearingRequired: PROCEDURE [Handle] RETURNS [BOOLEAN];

SetClearingRequired: PROCEDURE [window: Handle, required: BOOLEAN]
    RETURNS [old: BOOLEAN];

GetParent: PROCEDURE [Handle] RETURNS [Handle];

GetSibling: PROCEDURE [Handle] RETURNS [Handle];

GetSibling returns the next lower sibling of the argument window.

GetChild: PROCEDURE [Handle] RETURNS [Handle];

GetChild returns the topmost child of the argument window.

See also §59.2.4 for **Set** procedures that change a window's links to its parent, siblings, and child.

EntireBox: PROCEDURE [Handle] RETURNS [Box];

EntireBox returns the box [[0, 0], window.dims]. It is handy for invalidating the entire window.

GetBox: PROCEDURE [Handle] RETURNS [Box];

Note that there is no **SetBox**; **SlideAndSize** should be used instead.

GetPane: PROCEDURE [Handle] RETURNS [BOOLEAN];

GetPane returns whether or not the window is a window pane. The window pane property can only be set when the window is initialized.

IsCookieVariant: PROCEDURE [Handle] RETURNS [BOOLEAN];

Cookie cutters are not supported by the current release. **IsCookieVariant** should always return FALSE.

IsColorVariant: PROCEDURE [Handle] RETURNS [BOOLEAN];

Color is not supported by the current release. **IsColorVariant** should always return FALSE.

59.2.4 Window Tree and Window Box Manipulation

Basic operations are provided for constructing private trees from private windows and for inserting them into and removing them from the window tree. Other operations allow moving a window within a window tree and changing a window's location and size. Special operations are provided to perform common combinations of these operations.

Most clients obtain windows from some higher-level facilitiy like **FormWindow**; in such cases, the window typically has already been inserted into the window tree. Thus most clients will only use the following operations: **Stack**, **Slide**, **SlideAndStack**, **SlideAndSize**, **SlideAndSizeAndStack**.

Unless otherwise noted, all these operations may be applied either to windows in the window tree or to windows in a private tree. Operations performed on windows in private trees change tree links and the window's box but naturally create no invalid regions on the display.

As described in the overview, none of the operations in this section perform screen painting. They merely enqueue painting work to be performed by a later **Validate** operation.

IsDescendantOfRoot: PROCEDURE [Handle] RETURNS [BOOLEAN];

IsDescendantOfRoot returns TRUE if window is currently a part of the window tree.

ObscuredBySibling: PROCEDURE [Handle] RETURNS [BOOLEAN];

ObscuredBySibling returns TRUE if the box of any higher sibling intersects **window**'s box.

EnumerateTree: PROCEDURE [root: Handle, proc: PROCEDURE [window: Handle]];

EnumerateTree calls **proc** for every window in the tree rooted at **root**. The order of enumeration is not specified. Altering the tree while an enumeration is in progress causes unpredictable operation.

The following three operations allow constructing private trees from private windows.

SetParent: PROCEDURE [window, newParent: Handle] RETURNS [oldParent: Handle];

SetSibling: PROCEDURE [window, newSibling: Handle] RETURNS [oldSibling: Handle];

SetChild: PROCEDURE [window, newChild: Handle] RETURNS [oldChild: Handle];

These **Set** procedures set the parent, next lower sibling, or topmost child of **window**. No list manipulation nor consistency checking is done–these operations merely store their argument into the window object. If **window** is in the window tree, **Error[windowInTree]** is raised (**Stack**, et al. can be used in that case). If inconsistent calls to the **Set** procedures are made, **Error[windowNotChildOfParent]** is raised when some subsequent operation detects the inconsistency.

InsertIntoTree: PROCEDURE [window: Handle];

InsertIntoTree inserts a private window or subtree into any window tree. **window** is inserted as a child of **window.parent**. **window** is immediately above **window.sibling** in the sibling stack of the new parent; **window.sibling** = NIL makes it the bottom most sibling. **window.child** is the topmost child of a private tree that descends from the window--NIL if none. All of these fields of **window** may be set by using the **Set** procedures described above. The client can force painting of the windows just inserted by doing **window.GetParent[].ValidateTree[]**. **Error[noSuchSibling]** may be raised. Fine point: **InsertIntoTree** does not normally cause any painting activity. However, if a window that has a bitmap-under is inserted into the tree *and* the content of the bitmap is not available on the display, **ValidateTree** is done on that window's parent to obtain the content of the bitmap.

RemoveFromTree: PROCEDURE [Handle];

RemoveFromTree removes the window and all of its descendants from its containing tree. The window becomes the property of the client. The descendants of the window remain attached to it. The entire subtree may be later inserted back into a tree by using **InsertIntoTree**. The client can force painting of now-incorrect areas of the display by applying **ValidateTree** to any parent of the removed window. **Caution:** The **sibling** pointer of the removed window remains pointing to its former sibling in the tree. A client should take care that the **sibling** pointer of the window is set to the desired, valid in-tree sibling (or NIL) before doing a subsequent **InsertIntoTree**.

Stack: PROCEDURE [window: Handle, newSibling: Handle, newParent: Handle ← NIL];

Stack changes **window**'s location in its window tree, thus changing the window's depth in the apparent stack of windows on the screen. If **newParent** is not NIL, then **window** is moved to be a child of **newParent**; otherwise, its parent is unchanged. Next, the sibling stack then containing **window** is modified so that **window** is now immediately above **newSibling**, thus potentially obscuring siblings lower on its sibling stack. Supplying **newSibling** = NIL puts **window** on the bottom of the sibling stack. Unless **window** is already the top sibling, supplying **newSibling** = **window.GetParent.GetChild[]** puts **window** on the top of the stack. **Caution:** If **window** is the top sibling, the previous expression is a client error that is not guarded against. If one of **window** or **newParent** is in the window tree but the other is not, **Error[illegalStack]** is raised. **Error[noSuchSibling]** may also be raised.

Slide: PROCEDURE [window: Handle, newPlace: Place];

Slide changes **window**'s position relative to its parent. This procedure may be used to implement scrolling. **Error[whosSlidingRoot]** may be raised.

SlideAndStack: PROCEDURE [
window: Handle, newPlace: Place, newSibling: Handle, newParent: Handle ← NIL];

SlideAndStack performs a **Stack** and then a **Slide**, thus changing **window**'s location in its tree and its position within its new parent. **Error[illegalStack]**, **Error[noSuchSibling]**, and **Error[whosSlidingRoot]** may be raised.

Gravity: TYPE = {nil, nw, n, ne, e, se, s, sw, w, c, xxx};

Gravity indicates where the old pixel content of a window should go when it changes size. This allows **Window** to reuse any current window content that will be visible in its new configuration.

nil	The contents remain at their current *screen* position (not their window-relative position).
nw, n, ne, e, se, s, sw, w	The contents stay attached to the indicated compass point of the window, which is either a corner or the middle of a side; for example, nw means the contents stay in the upper-left corner.
c	The contents go in the middle of the new window–trimming or expansion occurs equally at opposite edges.
xxx	The contents are discarded.

SlideAndSize: PROCEDURE [window: Handle, newBox: Box, gravity: Gravity ← nw];

SlideAndSize changes both the location and size of **window**. **gravity** indicates what to do with the current contents of the window. **Error[sizingWithBitmapUnder]** and **Error[whosSlidingRoot]** may be raised.

SlideAndSizeAndStack: PROCEDURE [
window: Handle, newBox: Box, newSibling: Handle, newParent: Handle ← NIL,
gravity: Gravity ← nw];

SlideAndSizeAndStack performs a **Stack** and then a **SlideAndSize**, thus changing **window**'s location in its window tree and its position and size within its new parent. **Error[illegalStack]**, **Error[noSuchSibling]**, **Error[sizingWithBitmapUnder]**, and **Error[whosSlidingRoot]** may be raised.

SlideIconically: PROCEDURE [window: Handle, newPlace: Place];

SlideIconically is not implemented in the current release.

59.2.5 Causing Painting

A general description of painting is given in §59.1.3. The procedures below are used both to cause areas of the screen to be painted and actually to do the painting.

InvalidateBox: PROCEDURE [window: Handle, box: Box, clarity: Clarity ← isDirty];

Clarity: TYPE = {isClean, isDirty};

InvalidateBox declares that the current screen content of **box** in **window** is incorrect. **Window** adds **box** to the list of invalid regions of the window. **clarity** indicates the current state of the box. **clarity = isClean** means the region is already erased (all white); **isDirty**, that it contains some black. **Window** uses this information to avoid unnecessary clearing. **InvalidateBox** does not cause immediate display painting; only the **Validate** procedures do that. Note that a call on **InvalidateBox** followed by a call on **Validate** may result in no call to the display procedure--for example, if the invalidated area is not visible. If the window is not in the window tree, this operation does nothing.

Validate: PROCEDURE [window: Handle];

ValidateTree: PROCEDURE [window: Handle ← rootWindow];

Validate and **ValidateTree** are the only **Window** procedures that cause immediate display painting. Fine point: The few special cases that do not follow this rule are noted in the text. **Validate** acts only on **window**; **ValidateTree** acts on the tree whose root is **window**. Typically, a client updates its data structures and invalidate various regions. When the client is ready to have the display updated, one of the **Validate** procedures is called. If **window** is not in the window tree, this operation does nothing.

EnumerateInvalidBoxes: PROCEDURE [window: Handle, proc: PROCEDURE [Handle, Box]];

EnumerateInvalidBoxes is used within a window's display procedure to obtain the list of invalid regions of the window. **EnumerateInvalidBoxes** calls **proc** for each of the invalid boxes of **window**; **window** is passed to **proc** as its first argument. The second argument of **proc** describes the region that is invalid. **Note:** A display procedure need not worry about redundant painting outside the invalid regions; **Window** automatically discards the display procedure's paint that falls outside the invalid regions. This operation must only be called from within a display procedure, and **window** must be the window argument of the display procedure.

FreeBadPhosphorList: PROCEDURE [window: Handle];

In special circumstances, a display procedure may wish to paint into valid visible areas. **FreeBadPhosphorList** deletes the display procedure's invalid area list; for the lifetime of that invocation of that display procedure, paint requests are clipped only to the visible parts of the window. This operation must only be called from within a display procedure, and **window** must be the window argument of the display procedure.

TrimBoxStickouts: PROCEDURE [window: Handle, box: Box] RETURNS [Box];

TrimBoxStickouts returns a box that is the result of excluding any portion of **box** that sticks out of **window** or its ancestors. Display procedures may find it useful.

59.2.6 Errors

```
Error: ERROR [code: ErrorCode];

ErrorCode: TYPE = {
    illegalBitmap, illegalFloat, windowNotChildOfParent, whosSlidingRoot,
    noSuchSibling, noUnderVariant, windowInTree, sizingWithBitmapUnder,
    illegalStack, invalidParameter};
```

illegalBitmap	A window passed to **SetBitmapUnder** is not totally visible.
illegalFloat	See **Float**.
windowNotChildOfParent	A window is not in the list of its parent's children. This usually means that inconsistent calls to **SetParent**, **SetChild**, or **SetSibling** were made.

whosSlidingRoot	The client has attempted to move the root window.
noSuchSibling	An operation moving a window in the window tree specifies a new sibling that is not a child of the new parent.
noUnderVariant	A bitmap under operation was applied to a window that may not have a bitmap-under associated with it.
windowInTree	**SetParent**, **SetSibling**, or **SetChild** was applied to a window in the window tree. **Stack**, et al., can be used instead.
sizingWithBitmapUnder	A client has tried to change the size of a window that currently has a bitmap-under but does not have an AllocateUnderProc or a FreeUnderProc associated with it.
illegalStack	The client is attempting to move a window between parents, one of which is in the window tree and the other is not.
invalidParameter	The client has invoked an operation with invalid parameters.

59.2.7 Special Topic: Bitmap-Under

Bitmap-unders are described in §59.1.4. Most clients have no need for bitmap-unders.

IsBitmapUnderVariant: PROCEDURE [Handle] RETURNS [BOOLEAN];

IsBitmapUnderVariant returns TRUE if the window can be associated with a bitmap-under (that is, if **InitializeWindow[. . . , under: TRUE]**).

WordsForBitmapUnder: PROCEDURE [window: Handle] RETURNS [CARDINAL];

WordsForBitmapUnder returns the number of words of storage needed for a bitmap-under corresponding to the current size of **window**. **PagesForBitmapUnder** should be used rather than **WordsForBitmapUnder** because pages are a more appropriate unit to describe bitmap-unders and large bitmap-unders can exceed the number of words which can be returned from **WordsForBitmapUnder**.

PagesForBitmapUnder: PROCEDURE [window: Handle] RETURNS [LONG CARDINAL];

PagesForBitmapUnder returns the number of pages of storage needed for a bitmap-under corresponding to the current size of **window**. This procedure should be used rather than **WordsForBitmapUnder** because pages are a more appropriate unit to describe bitmap-unders and large bitmap-unders can exceed the number of words that can be returned from **WordsForBitmapUnder**. Fine Point: This procedure is currently exported through **WindowExtra**.

PagesForDims: PROCEDURE [dims:Dims] RETURNS [LONG CARDINAL];

PagesForDims returns the number of pages of storage needed for a bitmap-under corresponding to a window with dimensions of **dims**. This procedure is useful when a window has not yet been created, yet the size of the required bitmap is desired. This is useful when using **CreateBufferBacked** (see §59.2.9). Fine Point: This procedure is currently exported through **WindowExtra**.

```
SpecialSetBitmapUnder: PROCEDURE [
    window: Handle,
    allocateUnder: AllocateUnderProc,
    freeUnder: FreeUnderProc];
```

```
AllocateUnderProc: TYPE = PROCEDURE [pages: LONG CARDINAL] RETURNS [pointer: LONG
POINTER];
```

```
FreeUnderProc: TYPE = PROCEDURE [pointer: LONG POINTER];
```

SpecialSetBitmapUnder associates an **AllocateUnderProc** and a **FreeUnderProc** with **window**. **allocateUnder** is the client's procedure for allocating scratch storage area for **window**'s bitmap-under. **freeUnder** is the client's procedure for freeing **window**'s bitmap-under storage when it is no longer needed. **allocateUnder** and **freeUnder** are called during insertion and removal of window to and from the visible window tree as well as in sliding, stacking and moving of **window**.

Unlike **SetBitmapUnder**, **SpecialSetBitmapUnder** allows the Window implementation to get bitmap-under storage through the client's **AllocateUnderProc** during the life of the window and thus provides flexibility of bitmap-under storage size and allows the sizing of bitmap-under windows.

Note that the client is responsible for the actual allocation and freeing of space. The client's **AllocateUnderProc** should return NIL if there is not enough resources to provide **window** with a bitmap-under. If the **AllocateUnderProc** returns NIL, **window** ceases to have a bitmap-under until the next time the **AllocateUnderProc** is called and some space is returned.

If the window cannot be associated with a bitmap-under, **Error[noUnderVariant]** is raised. For **window** to be associated with a bitmap-under, it must be created through **New[...under: TRUE]** and **Initialize[...under:TRUE]**. Fine Point: This procedure is currently exported through **WindowExtra**.

```
GetAllocateUnderProc: PROCEDURE [window:Handle] RETURNS [allocateUnder:
AllocateUnderProc];
```

GetAllocateUnderProc returns the **AllocateUnderProc** associated with **window**. If **window** does not have an **AllocateUnderProc**, **GetAllocateUnderProc** returns NIL. Fine Point: This procedure is currently exported through **WindowExtra**.

```
GetFreeUnderProc: PROCEDURE [window:Handle] RETURNS [freeUnder: FreeUnderProc];
```

GetFreeUnderProc returns the **FreeUnderProc** associated with **window**. If **window** does not have a **FreeUnderProc**, **GetFreeUnderProc** returns NIL. Fine Point: This procedure is currently exported through **WindowExtra**.

```
SetBitmapUnder: PROCEDURE [
    window: Handle, pointer: LONG POINTER ← NIL,
    underChanged: UnderChangedProc ← NIL,
    mouseTransformer: MouseTransformerProc ← NIL] RETURNS [LONG POINTER];
```

```
UnderChangedProc: TYPE = PROCEDURE [Handle, Box];
```

```
MouseTransformerProc: TYPE = PROCEDURE [Handle, Place] RETURNS [Handle, Place];
```

SetBitmapUnder associates a bitmap-under with **window**. **pointer** describes a scratch storage area for the bitmap-under; its length must be as given by **WordsForBitmapUnder**. If **pointer** = NIL, the window ceases to have a bitmap-under. The pointer to any previous bitmap-under is returned; the client becomes the owner of that storage. The **underChanged** and **mouseTransformer** parameters are ignored in the current release. If the window cannot be associated with a bitmap-under, **Error[noUnderVariant]** is raised. If the window is in the window tree but is obscured by another window, **Error[illegalBitmap]** is raised. While the bitmap-under is in effect, the window's size cannot be changed; an attempt to do so will raise **Error[sizingWithBitmapUnder]**.

```
GetBitmapUnder: PROCEDURE [window: Handle] RETURNS [LONG POINTER];
```

GetBitmapUnder returns the pointer to the current bitmap-under for **window**; returns NIL if none. If the window cannot be associated with a bitmap-under, **Error[noUnderVariant]** is raised.

```
Float: PROCEDURE [window, temp: Handle, proc: FloatProc];
```

```
FloatProc: TYPE = PROCEDURE [window: Handle] RETURNS [place: Place, done: BOOLEAN];
```

Float moves a window continuously on the screen. **Float** first forces **window** to the top of its sibling stack, next does **ValidateTree[rootWindow]**, and then enters a loop for changing the window's position. In the loop, **Float** calls **proc**, passing **window** to it. If **proc** returns **done** = TRUE, the operation terminates and **Float** returns to the client. Otherwise, **Float** moves the window to **place** and repaints the display. It does so without calling any client display procedure; control returns to the top of the loop. The client must ensure that the window is wholly visible when moved to **place**. **temp** is used for temporary storage for the duration of the float operation. **temp** must be the same size as **window**, have a bitmap-under, and not be in the window tree. If **window** is not in the window tree, if **temp** *is* in the window tree, if either window lacks a bitmap-under, or if the windows have different sizes, **Error[illegalFloat]** is raised.

59.2.8 Special Topic: Linked Windows

When windows are linked together they form a set. Whatever is painted into one member of the set will be painted into all members of the set. Painting is clipped to the visible area for each window. The client will usually want to associate the same DisplayProcs and TIP.NotifyProcs with each window. Also, if these procedures use Context data then the same context should be set for each window. A window may be removed from its set by calling the **Unlink** procedure, or it will automatically be removed when the window is destroyed. Example 2 under Usage/Examples shows typical usage.

Link: PROCEDURE [new, after: Handle];

Link associates the **new** window with the **after** window. If **after** is already a member of a set of linked windows, then **new** will become a member of that set also. Otherwise, **new** and **after** will become a new set. A window may be a member of only one set. If **new** is already in a set, **Error[invalidParameter]** will be raised. Windows which are to be Linked are created just as any other window. Each set of linked windows are kept in a circularly linked list in the order specified by the client.This means that the **new** window will be inserted into the list following the **after** window. Fine Point: This procedure is currently exported through **WindowExtra**.

Unlink: PROCEDURE [window: Handle];

Unlink removes **window** from the set of linked windows it currently belongs to. If it is not a member of a set, then nothing happens. Fine Point: This procedure is currently exported through **WindowExtra**.

GetNextLink: PROCEDURE [window: Handle] RETURNS [next: Handle];

GetNextLink returns the next window in the set following **window**. If there are no windows linked to **window** then **NIL** will be returned. Because the windows are kept in a circular list, it is necessary for the client to check the window returned against the window first passed to **GetNextLink** in order to determine when all members of the list have been enumerated. This is demonstrated in Example 2. Fine Point: This procedure is currently exported through **WindowExtra**.

IsLink: PROCEDURE [window: Handle] RETURNS [yes: BOOLEAN];

IsLink returns returns **TRUE** if **window** is currently linked to any other window. Fine Point: This procedure is currently exported through **WindowExtra**.

59.2.9 Special Topic: Buffer Backed Windows

Buffer Backed windows provide a method of capturing the bitmaps created by procedures in the **Display** interface. This type of window must not be passed to any procedure in the **Window** interface except, **FreeBufferBacked**, **GetBox** and **EntireBox** or unpredictable results will occur. The client is responsible for flushing the backing buffer before painting into it - **Display.White** is one method. Example 3 under Usage/Examples shows a typical client.

```
CreateBufferBacked: PROCEDURE [
  dims: Dims, buffer: LONG POINTER,
  cookie, color: BOOLEAN ← FALSE, zone: UNCOUNTED ZONE← NIL]
  RETURNS [window: Handle, paintAddress: Environment.BitAddress, bpl: NATURAL];
```

CreateBufferBacked creates a window which will have dimensions corresponding to the **dims** parameter. The backing buffer pointed to by **buffer** must be of at least the size specified by **PagesForDims**. The first return value, **window**, can be passed to any procedure in the **Display** interface. The bits will be painted into the backing buffer rather than onto the display. The second value returned, **paintAddress**, points to the first bits in the buffer which will be painted. This may differ from the value of **buffer** for alignment reasons. The third value returned, **bpl**, specifies the bits per scan line. The **cookie**, **color** and **zone**

parameters have the same meaning as in the **New** and **Initialize** procedures. This window must be freed by calling **FreeBufferBacked** rather than **Free**. Fine Point: This procedure is currently exported through **WindowExtra**. See **Usage/Examples**.

FreeBufferBacked: PROCEDURE [**window**: **Handle**, **zone**: UNCOUNTED ZONE ← NIL];

FreeBufferBacked will free the window. The client must free the backing buffer. The **zone** parameter has the same meaning as in the **Free** procedure. Fine Point: This procedure is currently exported through **WindowExtra**.

59.3 Usage/Examples

A scrollbar is an example of a simple window. An entire **StarWindowShell** window is an example of a window that has many descendant windows--the window header, the scrollbars, and the main interior window used to display the content.

Window shields the client from interference between windows and from the presence of the edges of the display screen. A client can freely move a window around on or off the screen and alter its position in the stack of windows; **Window** automatically handles the overlapping.

Window automatically clips painting into windows to the visible interior of its parent window. A client can freely paint anywhere inside or even outside of its window as convenient.

It is always correct to paint more of a window than the minimum required. Simple clients may adopt a simple repaint strategy, invalidating and/or repainting a large part or even all of a window. Sophisticated clients may invalidate only the necessary parts of a window, thus allowing only small amounts of repainting and minimizing references to the window's backing data. This may result in improved performance.

A display procedure has available to it a list of invalid areas that need to be repainted. However, it may adopt the simple approach of ignoring this data and repainting the entire window. In any case, **Window** clips a display procedure's paint to the boundaries of the invalid regions.

Areas that project outside of a window's parent are trimmed for display purposes. Vertical scrolling can be implemented quite simply by embedding a tall *content window* in a short *clipping window* and then just **Slide**ing the position of the content window within the clipping window. Horizontal scrolling can be done in a similar way. The **StarWindowShell** interface supports this method of scrolling. This approach is limited by the domain of the coordinates, which are INTEGERS. Scrolling in this way is limited to +-2↑15 pixels offset from the frame window. If more scrolling than this is required, the client cannot use this technique, but must itself perform the transformation from data coordinates to window coordinates.

Since a window's location is defined in its parent window's coordinate system, moving a window automatically moves all of its descendant windows along with it.

Window itself has nothing to do with the keyboard and mouse. However, the **TIP** interface provides the facility for associating mouse and keyboard actions with a window.

59.3.1 Display Procedures and MONITORS

Any process may manipulate windows and thus cause screen painting activity. Even if one client always runs in the Notifier process, its window's display procedure may be called at any instant because of asynchronous activities by some other process. *If a window's display procedure uses any nonlocal variables in its painting activity (the usual case), those variables must be protected by a* **MONITOR**. Most display procedures are monitor entry procedures. Of course, if the display procedure only refers to immutable data, its operation need not be monitored.

Since a display procedure is usually a monitor entry, the client must avoid deadlocks by not invoking the display procedure from within other monitor procedures. This is the standard rule for monitors. Because calling **Validate** may cause **Window** to call the client's display procedure, calls to **Validate** must be done outside the client's monitor. The normal arrangement is (1) enter monitor, (2) update monitor data and **Invalidate** regions, (3) exit monitor, (4) **Validate** (which causes the display to be repainted).

59.3.2 Example 1

```
-- These excerpts are taken from <BWSHacks>1.0>Source>Puzzle15Impl.mesa

boxSize: CARDINAL = 32;
boxDims: Window.Dims = [boxSize, boxSize];

bodyWindowDims: Window.Dims = [boxSize*grid + 2, boxSize*grid + 2];
boxes: ARRAY [0..max) OF Window.Box;

MenuProc: MenuData.MenuProc = {
   rb: XString.ReaderBody ← XString.FromSTRING["15 Puzzle"L];
   shell: StarWindowShell.Handle = StarWindowShell.Create [name: @rb];
   -- Window.Initialize[] is called by StarWindowShell Impls.
   body: Window.Handle = StarWindowShell.CreateBody [
      sws: shell,
      box: [[0,0],bodyWindowDims],
      repaintProc: Redisplay,
      bodyNotifyProc: NotifyProc ];
   .
   .
   .
   .
   StarWindowShell.Push [shell];
   };

NotifyProc: TIP.NotifyProc = {
   data: Data ← LocalFind[window]; -- Use Context to find data for this instance.
   place: Window.Place;
   FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
      WITH z: input SELECT FROM
      coords = > place ← z.place;
      atom = > SELECT z.a FROM
         pointUp = > {
            box: CARDINAL ← ResolveToBox [place];
            IF Adjacent [data.empty, box] THEN {
                Window.InvalidateBox [window, boxes[data.empty]];
                Window.InvalidateBox [window, boxes[box]];
                SwapBoxWithEmpty [data, box];
                Window.Validate[window];
                };
            };
            ENDCASE;
        ENDCASE;
.
.
.

Redisplay: PROC [window: Window.Handle] = {
    -- This is the body window's display procedure.
    data: Data ← LocalFind[window]; -- Use Context to find data for this instance.
    vertical: Window.Dims ← [2, boxSize*grid];
    horizontal: Window.Dims ← [boxSize*grid, 2];
    place: Window.Place ← [0,0];
    -- Display the 15 numbers
    FOR i: CARDINAL IN [0..max) DO
        value: CARDINAL ← data.values[i];
        -- The bitmaps were created earlier (not shown in this example)
        Display.Bitmap [window, boxes[i], [@bitmaps[value],0,0], boxSize];
        ENDLOOP;
    -- Display the vertical lines
    FOR i: CARDINAL IN [0..grid + 2) DO
        Display.Black [window, [place,vertical]];
        place.x ← place.x + boxSize;
        ENDLOOP;
    -- Display the horizontal lines
    place ← [0,0];
    FOR i: CARDINAL IN [0..grid + 2) DO
        Display.Black [window, [place,horizontal]];
        place.y ← place.y + boxSize;
        ENDLOOP;
};

-- Register a command for invoking this tool
Init: PROC = {
    rb: XString.ReaderBody ← XString.FromSTRING["15 Puzzle"L];
    StarDesktop.AddItemToAttentionWindowMenu [
        MenuData.CreateItem [
            zone: Heap.systemZone,
            name: @rb,
            proc: MenuProc] ];
    .
    .
    .
    };
```

59.3.3 Example 2

```
-- This example demonstrates one simple use of Linked Windows.

-- How to Link windows
AddSplitWindow: PROC [currentWindow: Window.Handle] = {
   new: Window.Handle ←StarWindowShell.CreateBody[...]; -- Create a window
   displayProc: Window.DisplayProc ← Window.GetDisplayProc[currentWindow];
   notifyProc: TIP.NotifyProc ← TIP.GetNotifyProc[currentWindow];
   [] ← Window.SetDisplayProc[new, displayProc];
   [] ← TIP.SetNotifyProc[new, notifyProc];
   WindowExtra.Link[new, currentWindow];
   };

-- How to Unlink windows
RemoveSplitWindow: PROC [currentWindow: Window.Handle] = {
   [] ← Window.SetDisplayProc[currentWindow,NIL];
   [] ← TIP.SetNotifyProc[new,NIL];
   WindowExtra.Unlink[currentWindow];
   };

-- How to enumerate windows,
ActionProc: TYPE = PROCEDURE [w: Window.Handle];
EnumerateLinkedWindows: PROC [currentWindow: Window.Handle, proc: ActionProc]
 = {
   -- Start with 'currentWindow' and call 'proc' with each window.
   -- Note that we start with the window following currentWindow and
   -- finish with currentWindow.
   firstWindow, nextWindow: Window.Handle;
   IF NOT WindowExtra.IsLink[currentWindow] THEN RETURN;
   firstWindow ← WindowExtra.GetNextLink[currentWindow];
   FOR nextWindow ← firstWindow, WindowExtra.GetNextLink[nextWindow] DO
   proc[nextWindow];
   IF nextWindow = currentWindow THEN EXIT;
   ENDLOOP;
   };
```

59.3.4 Example 3

```
-- This is a very simple example of how to create an image in a Buffer Backed Window
-- and then paint it onto the screen. The image could be processed differently.

mainWindow: Window.Handle;
bbWindow: Window.Handle ← NIL;
paintAddress: Environment.BitAddress;
bpl: NATURAL;
dims: Window.Dims = [w: 800, h: 800]; -- For example
pages: LONG CARDINAL;
backingBuffer: LONG POINTER;
```

```
-- Create the window where the resulting bitmap will be painted.
-- Note: The bits could be copied to another source.
     mainWindow ← ...;

-- Allocate the buffer to back the window
 pages ← WindowExtra.PagesForDims[dims];
backingBuffer: LONG POINTER ← Space.ScratchMap[count: pages];

-- Create the Buffer Backed Window
[bbWindow, paintAddress, bpl] ← WindowExtra.CreateBufferBacked[
dims: dims, buffer: backingBuffer];

-- Clear the "window" (i.e. the buffer)
 Display.White[window:bbWindow, box: Window.GetBox[bbWindow]];

-- Paint something into the "window" (i.e. the buffer)
 Display.Black[
     window:, bbWindow
      box: ...];

-- Write the prepared image onto the screen inside 'mainWindow'
 Display.Bitmap[
     window: mainWindow,
      box: [...],
     address: paintAddress,
     bitmapBitWidth: bpl];

-- Cleanup
backingBuffer ← Space.Unmap[backingBuffer];
WindowExtra.FreeBufferBacked[bbWindow];
```

59.3.5 Example 4

-- This example demonstrates how to create, display and destroy a bitmap-under window.

```
CreateWindow: PROCEDURE RETURNS [window: Window.Handle] = {
     window ← Window.New[under:under];
     Window.Initialize[
          window: window,
          .
          .
          .
          under: under];

          -- if under is TRUE, window can later be associated with a bitmap under but does
          -- not necessarily have to if resources are not available.
          -- Needs to call both Window.New and Window.Initialize instead of
          -- Window.Create cuz the inline Window.Create does not call Initialize
          -- with the under parameter correctly.}; -
```

```
} -- CreateWindow

PushWindow: PROCEDURE[window: Window.Handle] = {
    IF Window.IsBitmapUnderVariant[window] THEN
    [] ← WindowExtra.SpecialSetBitmapUnder[
        window: window,
        allocateUnder: AllocateUnder,
         freeUnder:FreeUnder];
    Window.InsertIntoTree[window];
    Window.ValidateTree [];
}; -- PushWindow

DestroyWindow: PROCEDURE[window: Window: Handle] = {
    Window.RemoveFromTree[window];
    Window.Free[window];
    Window.ValidateTree[];
}; -- DestroyWindow

EnoughBackingFile: PROCEDURE [pagesForBU: LONG CARDINAL] RETURNS [BOOLEAN]
= {
    -- This procedure will return TRUE if we have enough scratch storage in Pilot's
    --anonymous backing file or in the extra backing file to make a bitmap-under.
    --Note this is only an approximation since returning TRUE does not mean that we
    --have enough contiguous space. Pilot might still have to do a File.Create.

    RETURN [SpacePerf.countDataPool - SpacePerf.currentUtilization > pagesForBU
    OR ExtraBackingFile.countDataPool - ExtraBackingFile.currentUtilization >
     pagesForBU];
}; -- EnoughBackingFile

AllocateUnder: WindowExtra.AllocateUnderProc = {
<<PROCEDURE [pages: LONG CARDINAL] RETURNS [pointer: LONG POINTER]>>
    ENABLE Space.InsufficientSpace, Volume.InsufficientSpace => CONTINUE;
    pointer ← NIL;
    IF EnoughBackingFile[pages] THEN pointer ← [Space.ScratchMap[pages]];
}; -- AllocateUnder

FreeUnder: WindowExtra.FreeUnderProc = {
<<PROCEDURE [pointer: LONG POINTER]>>
    [] ← Space.Unmap[pointer];
}; -- FreeUnder
```

59.4 Index of Interface Items

60

XChar

60.1 Overview

The **XChar** interface is part of a string package that supports the *Xerox Character Code Standard*, referred to in this document as the "standard." **XChar** defines the basic character type and some operations on it.

The standard defines 16-bit characters, which would permit up to 65,536 distinct characters. Reserving control character space reduces them to 35,532. It is convenient to partition the character code range into 256 blocks of 256 codes each. Each block is called a *character set*. This approach allows a convenient run-encoding scheme.

All the character sets currently defined are enumerated in **XCharSets**.

60.2 Interface Items

60.2.1 Character Representation

Character: TYPE = WORD;

Character is a 16-bit character.

Fine point: Currently only 16-bit characters are defined by the standard, but larger characters are not precluded. If the standard is extended to include more bits per character, the type **Character** will be redefined.

CharRep: TYPE = MACHINE DEPENDENT RECORD [set, code: Environment.Byte];

CharRep is a type that defines the representation of a character as character set and code. The operations **Code**, **Make**, and **Set** should be used instead of this type.

Code: PROCEDURE [c: Character] RETURNS [code: Environment.Byte] ;

Code returns the code within a character set of the character parameter.

Make: PROCEDURE [set, code: Environment.Byte] RETURNS [Character];

Make constructs a character, given a character set and a code within the character set.

Set: PROCEDURE [c: Character] RETURNS [set: Environment.Byte] ;

Set returns the character set of the character parameter.

null: Character = 0;
not: Character = 177777B;

not is a value that may be used by operations that return a character to signify that no characters remain.

60.2.2 JoinDirection and StreakNature

JoinDirection: TYPE = {nextCharToLeft, nextCharToRight};

JoinDirection specifies whether a character goes left to right or right to left.

GetJoinDirection: PROCEDURE [Character] RETURNS [JoinDirection];

GetJoinDirection returns the join direction for a character, given its set and code within its set.

ArabicFirstRightToLeftCharCode: Environment.Byte = 60B;

ArabicFirstRightToLeftCharCode is used by **GetJoinDirection**.

StreakNature: TYPE = {leftToRight, rightToLeft};

GetStreakNature: PROCEDURE [Character] RETURNS [StreakNature];

Returns a characters **StreakNature** (see **SimpleTextDisplay.StreakSuccession**).

60.2.3 Case

Decase: PROCEDURE [c: Character] RETURNS [Character];

Decase is a case-stripping operation. It returns **c** with all case information removed. This is useful when comparing characters with case ignored. Only characters in character sets zero (Latin), 46(Greek), and 47(Cyrillic) are affected.

LowerCase: PROCEDURE [c: Character] RETURNS [Character];

LowerCase returns the lowercase representation of the character **c**. Only characters in character set zero (Latin), 46 (Greek), and 47 (Cyrillic) are affected.

UpperCase: PROCEDURE [c: Character] RETURNS [Character];

UpperCase returns the uppercase representation of the character **c**. Only characters in character set zero (Latin), 46 (Greek), and 47 (Cyrillic) are affected.

60.3 Usage/Examples

The following two examples create specific characters. XChar.Make is also useful if the character set and code are not known at compile time, but are known at run time.

60.3.1 Creating an ASCII Character

The following example creates an ASCII CR character.

```
c: XChar.Character ← XChar.Make[set: XCharSets.Sets.latin.ORD, code:LOOPHOLE[Ascii.CR]];
```

60.3.2 Creating a Greek Character

The following example creates an α from the Greek character set.

```
c: XChar.Character ← XChar.Make[set:XCharSets.Sets.greek.ORD, code:
XCharSet46.Codes46.lowerAlpha.ORD];
```

60.4 Index of Interface Items

61

XCharSets, XCharSetNNN

61.1 Overview

XCharSets enumerates the character sets defined in the *Xerox Character Code Standard.* This chapter also describes a collection of interfaces that enumerate the character codes of several common character sets. This collection of interfaces is **XCharSetNNN**.

61.2 Interface Items

61.2.1 Sets

```
Sets: TYPE = MACHINE DEPENDENT {
   latin(0), firstUnused1(1), lastUnused1(40B), jisSymbol1(41B), jisSymbol2(42B),
   extendedLatin(43B), hiragana(44B), katakana(45B), greek(46B), cyrillic(47B),
   firstUserKanji1(50B), lastUserKanji1(57B), firstLevel1Kanji(60B), lastLevel1Kanji(117B),
   firstLevel2Kanji(120B), lastLevel2Kanji(163B), jSymbol3(164B), firstUserKanji2(165B),
   lastUserKanji2(176B), firstUnused2(177B), lastUnused2(240B), firstReserved1(241B),
   lastReserved1(337B), arabic(340B), hebrew(341B), firstReserved2(342B),
   lastReserved2(355B), generalSymbols2(356B), generalSymbols1(357B),
   firstRendering(360B), lastRendering(375B), userDefined(376B), selectCode(377B)};
```

Sets enumerates the character sets. Specific character sets have values defined, such as **Latin** and **Hiragana**. Character set families such as **Kanji** and unused or reserved portions of the character set enumeration are specified by first and last values; for example, **firstUserKanji1** and **lastUserKanji1**.

For those eleven character sets whose codes are specified in the standard, an interface has been defined that contains an enumerated type enumerating the codes within the character set and a Make procedure that makes a character, given a code literal.

For example, the interface **XCharSet356** has the following definitions:

```
Make: PROCEDURE [code: Codes356] RETURNS [Character];
```

```
Codes356: TYPE = MACHINE DEPENDENT {
  thickSpace(41B), fourEmSpace(42B), hairSpace(43B), punctuationSpace(44B),
  decimalPoint(56B), absoluteValue(174B), similarTo(176B), escape(377B)};
```

61.2.2 Enumeration of Character Sets

Table 61.1 enumerates the eleven character sets whose codes are specified in the standard, the interface in which they are contained, and the enumerated type name for that interface.

Character Set	Interface	Enumerated Type
Latin	XCharSet0	Codes0
jisSymbol1	XCharSet41	Codes41
jisSymbol2	XCharSet42	Codes42
extendedLatin	XCharSet43	Codes43
Hiragana	XCharSet44	Codes44
Katakana	XCharSet45	Codes45
Greek	XCharSet46	Codes46
Cyrillic	XCharSet47	Codes47
jSymbol3	XCharSet164	Codes164
arabic	XCharSet340	Codes340
hebrew	XCharSet341	Codes341
generalSymbols2	XCharSet356	Codes356
generalSymbols1	XCharSet357	Codes357
firstRendering	XCharSet360	Codes360
accentedLatin	XCharSet361	Codes361

Table 61.1: Standard Character Sets

61.3 Usage/Examples

61.3.1 Creating a Greek Character

The following example shows two ways to create an α from the Greek character set.

```
c: XChar.Character ← XChar.Make[set:XCharSets.Sets.greek.ORD, code:
XCharSet46.Codes46.lowerAlpha.ORD];

c: XChar.Character ← XCharSet46.Make[code: XCharSet46.Codes46.lowerAlpha];
```

61.4 Index of Interface Items

62 XComSoftMessage

62.1 Overview

This interface assigns the *global handle* and message keys for all the messages the system requires for system templates (such as time and date formatting, numbers, and so forth.). The **XMessage** interface deals with system messages; it must be understood before using this interface.

62.2 Interface Items

62.2.1 Obtaining Message Handle

```
GetHandle: PROCEDURE RETURNS [h: XMessage.Handle];
```

This procedure returns a handle for system-required messages that have already been initialized and allocated, and registered by the **XComSoftMessage** implementation.

62.2.2 Message Keys

```
Keys: TYPE = MACHINE DEPENDENT {
   time(0), date(1), dateAndTime(2), am(3), pm(4), january(5), february(6), march(7),
   april(8), may(9), june(10), july(11), august(12), september(13), october(14),
   november(15), december(16), monday(17), tuesday(18), wednesday(19), thursday(20),
   friday(21), saturday(22), sunday(23), decimalSeparator(24), thousandsSeparator(25)};
```

time, **date**, and **dateAndTime** are available through the **XTime** interface; they may be used as templates in calls to **XTime.ParseReader** or **XTime.Append**.

```
Months: TYPE = Keys [january..december];
```

```
DaysOfWeek: TYPE = Keys [monday..sunday];
```

62.3 Usage/Examples

```
OPEN XCSM: XComSoftMessage;

systemMsgs: XMessage.Handle ← XCSM.GetHandle [];
mondayString: XString.Reader ← XMessage.Get [
    systemMsgs, XCSM.DaysOfWeek.monday.ORD];
```

62.4 Index of Interface Items

63

XDigits

63.1 Overview

The **XDigits** interface provides the definitions and procedures for implementing and manipulating different number representations.

The average client will only use **XDigits.NumberParms** that is provided as a parameter for certain input/output number routines such as **XStringX2.ReaderToNumberX** and **XFormatX.NumberX**.

63.2 Interface Items

63.2.1 Representation

```
Representation:TYPE = MACHINE DEPENDENT {
   default(0), ascii(1), arabian(2), persian(3), urdu(4), Jkanji(5), Chanzi(6),
   Khanja(7), amharic(8), burmese(9), khmer(10), lao(11), thai(12), devanagari(13),
   bengali(14), firstFree(15), last(255)};
```

Representation lists all known number representations.

```
Digit: TYPE = [0..9];
```

```
Symbols: TYPE = LONG DESCRIPTOR FOR ARRAY Digit OF XChar.Character;
```

Each number representation will have **Symbols** associated with it.

```
NumberParms: TYPE = RECORD [
   representation: Representation,
   parensForNeg: BOOLEAN,
   thousandsSep: XChar.Character];
```

```
defaultNumberParms: NumberParms = [default, FALSE, XChar.not];
```

NumberParms is the parameter for several number input/output routines. **representation** is used to select the symbols to be used to input or output the digits. If **representation** is defaulted, the default representation specified in the User Profile is used. If **parensForNeg**

is TRUE, numbers within parentheses will be accepted and negated on input. On output, if **parensForNeg** is TRUE, negative numbers will be emitted within parentheses instead of the minus sign. **thousandsSep** will be permitted on input provided it is not the first or last character. No other checks are made on input. On output, **thousandsSep** will be emitted every three digits. **NumberParms** is currently a parameter in **XStringX2.ReaderToNumberX**, **XTokenX.NumberX**, **XTokenX.DecimalX**, **XFormatX.NumberFormatX**, **XFormatX.NumberX**, **XFormatX.DecimalX**, **XLRealX2.ReaderToNumberX**, **XLRealX2.ReadNumberX** and **XLRealX2.FormatNumberX**.

63.2.2 Operations

```
GetSymbols: PROCEDURE [representation: Representation]
  RETURNS [symbols: Symbols];
```

GetSymbols returns the **symbols** associated with **representation**. If **representation** does not have any symbols associated with it, NIL will be returned.

```
SetSymbols: PROCEDURE [
  representation: Representation[ascii..last],
  symbols: Symbols,
  label: XString.Reader ← NIL] ;
```

SetsSymbols sets the **symbols** for **representation** . If **label** is non-NIL, it will replace the built-in label for the **representation**. **label** will copied and storage for the bytes will be allocated out of a private zone.

```
SetDefault: PROCEDURE [representation: Representation[ascii..last]];
```

SetDefault sets the default representation to be **representation**.

```
GetLabel: PROCEDURE [representation: Representation] RETURNS [label: XString.Reader];
```

GetLabel returns the **label** for **representation**. NIL is returned if there is no **label** for the **representation**. **GetLabel** does not copy the **label** but returns a pointer to it. It should not be changed by the client.

63.3 Usage/Examples

63.3.1 Assigning symbols

Clients adding a digit representation that has not been implemented will want to use **SetSymbols**.

```
AddArabianSymbols: PROCEDURE = {
  arabianSymbols: ARRAY Digit OF XChar.Character ← [
    XCharSet340.Make[indian0],
    XCharSet340.Make[indian1],
    XCharSet340.Make[indian2],
    XCharSet340.Make[indian3],
    XCharSet340.Make[indian4],
    XCharSet340.Make[indian5],
    XCharSet340.Make[indian6],
    XCharSet340.Make[indian7],
    XCharSet340.Make[indian8],
    XCharSet340.Make[indian9]];
SetSymbols[representation: arabian, symbols: DESCRIPTOR[arabianSymbols]];};
```

63.4 Index of Interface Items

64

XFormat

64.1 Overview

The **XFormat** package provides procedures for formatting various types into **XString.Readers**. The procedures require the client to supply an output procedure and a piece of data to be formatted. Where appropriate, a format specification is also required.

64.1.1 Major Data Structures

The major data structure is the **Handle**,which points to an object containing a **FormatProc**, an **XString.Context**,and some **ClientData**. All the formatting operations take a handle as the destination of the formatted character string. The **FormatProc** is the main component of an **Object**. It should pass the characters of its reader parameter to the output sink it implements and update the object's context to reflect the context of the last character of the reader parameter.

The other major data structure is the **NumberFormat**, which defines how numbers are to be converted to text strings. It includes the base of the number, the number of columns the text string should contain, whether to treat the number as signed or unsigned, and whether to fill leading columns with zeros or spaces.

A **FormatProc** is the destination of all output from the format routines. It is the main component of an **Object**. It should pass the characters of **r** to the appropriate sink and update **h.context** to reflect the context of the last character of **r**.

64.1.2 Operations

There are two major classes of operations in **XFormat**. The first class is used to format various data types and pass them to a format procedure. These operations contain simple text operations such as **Blanks**, **Reader**, and **String**; numeric operations such as **Decimal** and **Number**; network-related operations such as **NetworkAddress** and **HostNumber**; and some compatibility routines such as **NSString**. All these operations direct their output to the format procedure in their **handle** parameter. If this parameter is defaulted, it is directed to the default output sink.

The second class of operations provide built-in format procedures that direct their output to the following well-known data types: XString.Writer, Stream.Handle, TTY.Handle, and NSString.String.

64.2 Interface Items

64.2.1 Handles and Objects

```
Handle: TYPE = LONG POINTER TO Object;

Object: TYPE = RECORD [
  proc: FormatProc,
  context: XString.Context ← XString.vanillaContext,
  data: ClientData ← NIL];

FormatProc:TYPE = PROCEDURE [r: XString.Reader, h: Handle];

ClientData: TYPE = LONG POINTER;
```

A Handle is a parameter to all the formatting operations. Its object encapsulates the output sink that is the destination of all formatted text. The proc field is called one or more times for each formatting operation; it should pass the characters of its reader parameter to the output sink it implements. The context field is used to hold the context of the last character sent to the format procedure. It should be updated by the format procedure. The data field allows client-specific information to be passed to the format procedure.

64.2.2 Default Output Sink

```
SetDefaultOutputSink: PROCEDURE [new: Object] RETURNS [old: Object];
```

SetDefaultOutputSink sets the default object that is the default destination for all formatted output. For each of the formatting operations, if the handle parameter is NIL, it is directed to the default output sink. The default output sink is initialized to an object that ignores all results.

64.2.3 Text Operations

```
Blanks: PROCEDURE [h: Handle ← NIL, n: CARDINAL ← 1];
```

Blanks calls on h.proc with readers that contain a total of n blanks. h.proc may be called more than once.

```
Block: PROCEDURE [h: Handle ← NIL, block: Environment.Block];
```

Block calls on h.proc with a reader that contains the characters in block.

```
Char: PROCEDURE [h: Handle ← NIL, char: XString.Character];
```

Char calls on h.proc with a reader that contains only the character char.

CR: PROCEDURE [h: Handle ← NIL, n: CARDINAL ← 1];

CR calls on **h.proc** with readers that contain a total of **n** carriage returns (15C). **h.proc** may be called more than once.

Line: PROCEDURE [h: Handle ← NIL, r: XString.Reader, n: CARDINAL ← 1];

Line calls on **h.proc** with **r** and then readers that contain a total of **n** carriage returns (15C). **h.proc** will be called more than once.

Reader: PROCEDURE [h: Handle ← NIL, r: XString.Reader];

Reader calls on **h.proc** with **r**.

ReaderBody: PROCEDURE [h: Handle ← NIL, rb: XString.Reader];

ReaderBody calls on **h.proc** with **@rb**.

String: PROCEDURE [h: Handle ← NIL, s: LONG STRING];

String calls on **h.proc** with readers that contain the characters in **s**.

64.2.4 Number Formats

```
NumberFormat: TYPE = RECORD [base: [2..36] ← 10,
  zerofill: BOOLEAN ← FALSE, signed: BOOLEAN ← FALSE, columns: [0..255] ← 0 ];
```

NumberFormat is used by the number-formatting procedures. The number will be formatted in base **base** in a field at least **columns** wide (zero means "use as many as needed"). If **zerofill** is TRUE, the extra columns are filled with zeros; otherwise, spaces are used. If **signed** is TRUE and the number is less than zero, a minus sign preceeds all output, except for columns that are filled with spaces. For bases greater than 10, the characters 'A..'Z are used as digits.

```
NumberFormatX: TYPE = RECORD [base: [2..36] ← 10,
   zerofill: BOOLEAN ← FALSE, signed: BOOLEAN ← FALSE, columns: [0..255] ← 0,
   numberParms: XDigits.NumberParms ← XDigits.defaultNumberParms];
```

NumberFormatX is the same as **NumberFormat** except that **numberParms** can be specified. If **base** is other than 10, **numberParms.representation** and **numberParms.thousandsSep** are ignored. See the XDigits chapter for more details. Fine Point: This type is currently found in **XFormatX**.

```
DecimalFormat: NumberFormat = [
  base: 10, zerofill: FALSE, signed: TRUE, columns: 0];

HexFormat: NumberFormat = [
  base: 16, zerofill: FALSE, signed: FALSE, columns: 0];

OctalFormat: NumberFormat = [
  base: 8, zerofill: FALSE, signed: FALSE, columns: 0];
```

UnsignedDecimalFormat: NumberFormat = [
  base: 10, zerofill: FALSE, signed: FALSE, columns: 0];

These are useful number format constants. The output will fill as many columns as needed.

64.2.5 Numeric Operations

Number: PROCEDURE [
  h: Handle ← NIL, n: LONG UNSPECIFIED, format: NumberFormat];

Number formats **n** to a string according to the number format **format**. The number will be formatted in base **base** in a field at least **columns** wide (zero means "use as many as needed"). If **zerofill** is **TRUE**, the extra columns are filled with zeros; otherwise, spaces are used. If **signed** is **TRUE** and the number is less than zero, a minus sign preceeds all output, except for columns that are filled with spaces. For bases greater than 10, the characters 'A..'Z are used as digits. **h.proc** will be called several times with pieces of the output as they are generated.

NumberX: PROCEDURE [
  h: Handle ← NIL, n: LONG UNSPECIFIED, format: NumberFormatX];

NumberX formats **n** to a string according to the number format **format**. **NumberX** is the same as **Number** except that **format** is of type **NumberFormatX** wherein the **numberParms** can be specified. See the XDigits chapter for more details. Fine Point: This procedure is currently exported through **XFormatX**.

Decimal: PROCEDURE [h: Handle ← NIL, n: LONG INTEGER];

Decimal converts **n** to signed base 10. It is equivalent to **Number[h, n, DecimalFormat]**.

DecimalX: PROCEDURE [h: Handle ← NIL, n: LONG INTEGER,
  numberParms: XDigit.NumberParms ←XDigits.defaultNumberParms];

DecimalX converts **n** to signed base 10. It is equivalent to **NumberX[h, n, DecimalFormat]**. See the XDigits chapter for more details. Fine Point: This procedure is currently exported through **XFormatX**.

Hex: PROCEDURE [h: Handle ← NIL, n: LONG CARDINAL];

Hex converts **n** to signed base 16. It is equivalent to **Number[h, n, HexFormat]**.

Octal: PROCEDURE [h: Handle ← NIL, n: LONG UNSPECIFIED];

Octal convert **n** to base 8. When **n** is greater than 7, the character 'B is appended. It is equivalent to **Number[h, n, OctalFormat]; IF n > 7 THEN Char[h, 'B.ORD]**.

64.2.6 Built-in Sinks

The **XFormat** interface provides several built-in format procedures that know how to send output to particular destinations. For each of the four known types of destinations (**XString.Writer**, **Stream.Handle**, **TTY.Handle**, and **NSString.String**), there are both the format procedure as well as an operation that returns an object initialized with the appropriate format procedure and destination data. Both the format procedures and the object operations may raise the error Error[nilData] if the expected data is NIL.

NSStringProc: FormatProc;

NSStringObject: PROCEDURE [s: LONG POINTER TO NSString.String] RETURNS [Object];

NSStringProc appends the reader to an NSString.String. It expects h.data to be a LONG POINTER TO NSString.String. NSStringObject constructs an object whose proc is NSStringProc and whose data is s.

StreamProc: FormatProc;

StreamObject: PROCEDURE [sH: Stream.Handle] RETURNS [Object];

StreamProc puts the bytes of the reader to a Stream.Handle. It expects h.data to be a Stream.Handle. StreamObject constructs an object whose proc is StreamProc and whose data is sH.

TTYProc: FormatProc;

TTYObject: PROCEDURE [h: TTY.Handle] RETURNS [Object];

TTYProc puts the bytes of the reader to a TTY.Handle. It expects h.data to be a TTY.Handle. TTYObject constructs an object whose proc is TTYProc and whose data is h.

WriterProc: FormatProc;

WriterObject: PROCEDURE [w: XString.Writer] RETURNS [Object];

WriterProc appends the reader to a XString.Writer. It expects h.data to be a XString.Writer. WriterObject constructs an object whose proc is WriterProc and whose data is w.

64.2.7 Date Operation

DateFormat: TYPE = {dateOnly, timeOnly, dateAndTime};

DateFormat allows the user to specify which template from XTime is used when the date is to be formatted by the procedure Date.

```
Date: PROCEDURE [
   h: Handle ← NIL, time: System.GreenwichMeanTime ← System.gmtEpoch,
   format: DateFormat ← dateAndTime];
```

Date converts time to a string by calling XTime.Append, using format to specify which template to use. h.proc is then called. If time is defaulted, the current time is used.

64.2.8 Network Data Operations

NetFormat: TYPE = {octal, hex, productSoftware};

NetFormat is used by the procedures that format network addresses. **octal** converts the number to octal, **hex** converts to hex, and **productSoftware** converts the item to a decimal number and then inserts a "-" every three characters, starting from the right. An example of a number in product software format is 4-294-967-295.

```
HostNumber: PROCEDURE [
  h: Handle ← NIL, hostNumber: System.HostNumber, format: NetFormat];
```

HostNumber calls on **h.proc** with a reader that contains **hostNumber** formatted as defined by **format**.

```
NetworkAddress: PROCEDURE
   h: Handle ← NIL, networkAddress: System.NetworkAddress, format: NetFormat];
```

NetworkAddress calls on **h.proc** with a reader that contains **networkAddress** with the form *network-number#host-number#socket-number*, where the format of the various components isdetermined by **format**.

```
NetworkNumber: PROCEDURE [
   h: Handle ← NIL, networkNumber: System.NetworkNumber, format: NetFormat];
```

NetworkNumber calls on **h.proc** with a reader that contains **networkNumber** formatted as defined by **format**.

```
SocketNumber: PROCEDURE [
   h: Handle ← NIL, socketNumber: System.SocketNumber, format: NetFormat];
```

SocketNumber calls on **h.proc** with a reader that contains **socketNumber** formatted as defined by **format**.

64.2.9 NSString Operations

```
NSChar: PROCEDURE [h: Handle ← NIL, char: NSString.Character];
```

NSChar calls on **h.proc** with a reader that contains the character **char**.

```
NSLine: PROCEDURE [h: Handle ← NIL, s: NSString.String, n: CARDINAL ← 1];
```

NSLine calls on **h.proc** with a reader that contains the characters in **s**, then calls on readers that contain a total of **n** carriage returns (15C). **h.proc** may be called more than once.

```
NSString: PROCEDURE [h: Handle ← NIL, s: NSString.String];
```

NSString calls on **h.proc** with a reader that contains the characters in **s**.

64.2.10 Errors

```
Error: ERROR [code: ErrorCode] ;

ErrorCode: TYPE = {invalidFormat, nilData};
```

invalidFormat	The term **invalidFormat** means an invalid operation has been attempted
nilData	The term **nilData** means **h.data** was NIL, but the format procedures wanted valid data.

64.3 Usage/Examples

64.3.1 Using Built-in Sinks

The **XFormat** interface allows clients to convert data types to their textual representation. By using the built-in sinks, clients can put this text into streams, **tty.handle**, and append to writers. In particular, although the **XString** interface does not include any append number operations, **XFormat** may be used to accomplish this task.

```
AppendNumber: PROCEDURE [
  w: XString.Writer, n: LONG INTEGER, format: XFormat.NumberFormat] = {
  xfo: XFormat.Object ← XFormat.WriterObject[w];
  XFormat.Number[h: @xfo, n: n, format: format]};
```

64.3.2 Creating New Format Procedures

While **XFormat** provides some useful output sinks, clients may wish to build new sinks. The following example hypothesizes a log window that can display text in a window and allows appending of text to the end.

```
LogWindow: DEFINITIONS = {
  Create: PROCEDURE [w: Window.Handle, file: NSFile.Handle];
  Destroy: PROCEDURE [w: Window.Handle];

  LogReader: PROCEDURE [w: Window.Handle, r: XString.Reader];
  Info: PROCEDURE [w: Window.Handle] RETURNS [
    file: NSFile.Handle, nChars: LONG INTEGER, endContext: XString.Context];

  LogFormatProc: XFormat.FormatProc;
  LogFormatObject: PROCEDURE [w: Window.Handle] RETURNS [object: XFormat.Object]

  ErrorCode: TYPE = {notALogWindow};
  Error: Error [code: ErrorCode];
  }..

LogWindowImpl: PROGRAM = {

  Create: PUBLIC PROCEDURE [w: Window.Handle, file: NSFile.Handle] = {...};
  Destroy: PUBLIC PROCEDURE [w: Window.Handle] = {...};

  LogReader: PUBLIC PROCEDURE [w: Window.Handle, r: XString.Reader] = {...};
  Info: PUBLIC PROCEDURE [w: Window.Handle] RETURNS [
    file: NSFile.Handle, nChars: LONG INTEGER, endContext: XString.Context] = {...};

  LogFormatProc: PUBLIC XFormat.FormatProc = {
    w: Window.Handle = h.data;
    IF w = NIL THEN ERROR XFormat.Error[nilData];
    LogReader[w: w, r: r];
    h.context ← Info[w].endContext};

  LogFormatObject: PUBLIC PROCEDURE [
    w: Window.Handle] RETURNS [object: XFormat.Object] = {
      IF w = NIL THEN ERROR XFormat.Error[nilData];
      RETURN[[proc: LogFormatProc, context: Info[w].endContext, data: w]]};

  }..
```

The bulk of the work is done in the **LogReader** procedure. It is assumed that the log window keeps track of the context of the end of the log so that it will add the necessary character set shift information when a reader that begins with a different character set is logged. If the log window didn't take care of this, the format procedure would have to set that itself, as the stream format procedure example below shows.

```
StreamProc: PUBLIC XFormat.FormatProc = {
  stream: Stream.Handle = h.data;
  startsWith377B: BOOLEAN;
  c: XString.Context;
  IF stream = NIL THEN ERROR XFormat.Error[nilData];
  [context: c, startsWith377B: startsWith377B] ← XString.ReaderInfo[r];
  SELECT TRUE FROM
    startsWith377B => NULL;
    c.suffixSize = 2 =>
      IF h.context.suffixSize = 1 THEN {
        stream.PutByte[377B]; stream.PutByte[377B]; stream.PutByte[0]};
    h.context.suffixSize = 2, c.prefix # h.context.prefix => {
      stream.PutByte[377B]; stream.PutByte[c.prefix]};
    ENDCASE;
  stream.PutBlock[block: XString.Block[r]];
  h.context ← XString.ComputeEndContext[r]};
```

64.4 Index of Interface Items

65

XLReal

65.1 Overview

XLReal is a decimal real package that supports manipulation of real numbers with greater precision than Mesa REALs.

65.2 Interface Items

65.2.1 Representation

Numbers are maintained as 13 decimal digits of signed mantissa with a 10-bit exponent (-512 to 511). All routines maintain the normalized numbers, i.e., the first digit is non-zero. The assumed decimal point is after the first digit. Numbers are stored as opaque objects occupying 4 words (64 bits).

```
Digit: TYPE = [0..9];
Number: TYPE [4];
Bits: TYPE = ARRAY [0..4) OF CARDINAL;
ValidExponent: TYPE = [-512..511];
Digits: TYPE = PACKED ARRAY [0..accuracy) OF Digit;
accuracy: NATURAL = 13;
```

65.2.2 Conversion

XLReal provides routines to convert numbers to and from other representations such as LONG INTEGERs and REALs as well as routines to look at pieces of numbers.

```
NumberToPair: PROCEDURE [
  n: Number, digits: [1..accuracy]] RETURNS [negative: BOOLEAN, exp: INTEGER, mantissa:
  Digits];

PairToNumber: PROCEDURE[
  negative: BOOLEAN, exp:INTEGER, mantissa: Digits] RETURNS [n: Number];
```

In **PairToNumber** and **NumberToPair**, the decimal point. is between mantissa[0] and mantissa[1]. **NumberToPair** rounds **n** so that **mantissa** contains **digits** significant digits. All other digits are zero. **NumberToPair** may raise **Error[notANumber]**. **PairToNumber** may raise **Error[underflow]**.

IntegerPart, FractionPart: PROCEDURE [Number] RETURNS [Number];

IntegerPart and **FractionPart** may raise **Error[notANumber]**. **FractionPart** may also raise **Error[overflow]**.

Fix:PROCEDURE [Number] RETURNS [LONG INTEGER];

Float: PROCEDURE [LONG INTEGER] RETURNS [Number];

Fix may raise **Error[notANumber]** and **Error[overflow]**. **Fix** rounds (as opposed to truncate).

ToREAL:PROCEDURE [number: Number] RETURNS [REAL];

ToREAL converts **number** to a **REAL**. If **Abs[number]** is greater than the largest **REAL** (3.40282347E38) it will return **Real.PlusInfinity** or **Real.MinusInfinity** depending on **number**'s sign. If it less than the smallest **REAL** (1.17549435E-38) it will return **Real.PlusZero** or **Real.MinusZero** depending on **number's** sign. Some precision may be lost since a **number's** precision is greater than a **REAL**'s precision. **ToREAL** may raise **Error[notANumber]**. Fine Point: This procedure is currently exported through **XLRealX**.

FromREAL: PROCEDURE [real: REAL] RETURNS [Number];

FromREAL converts **real** to a **Number**. If **real** is **Real.PlusInfinity** or **Real.MinusInfinity**, **FromREAL** returns 9.999999999999E511 or -9.999999999999E511 depending on **real's** sign. If **real** is not a number, **FromREAL** returns **MakeSpecial[0]**;. Fine Point: This procedure is currently exported through **XLRealX**.

65.2.3 Input/Output

The input and output routines convert numbers to and from a stream of characters.

ReaderToNumber: PROCEDURE[r: XString.Reader] RETURNS [Number];

ReaderToNumber converts **r** into a number. The number may have leading and trailing white space (spaces, tabs and returns). It may raise **Error[overflow]** if the number is too big or **Error[notANumber]** if the reader contains invalid characters.

```
ReaderToNumberX: PROCEDURE[
   r: XString.Reader,
   numberParms: XDigits.NumberParms ←XDigits.defaultNumberParms,
   decimalSep: XChar.Character ←XChar.not] RETURNS [Number];
```

ReaderToNumberX is the same as **ReaderToNumber** except that the **numberParms** and **decimalSep** can be specified. If **decimalSep** is defaulted then the decimal separator described in the XComSoft messages file is used. See the **XDigits** chapter for more information. Fine Point: This procedure is currently exported through **XLRealX2**.

```
ReadNumber: PROCEDURE [
   get: PROC RETURNS [XChar.Character], putback: PROC [XChar.Character]] RETURNS [Number];
```

ReadNumber converts the stream of characters from **get** into a number. Any character other than white space, '+, '–, 'E, 'e, '0-'9, or the decimal separator will cause conversion to terminate and **putback** will be called with that character. It may raise **Error[overflow]**.

```
ReadNumberX: PROCEDURE [
   get: PROC RETURNS [XChar.Character],
   putback: PROC [XChar.Character],
   numberParms: XDigits.NumberParms ← XDigits.defaultNumberParms,
   decimalSep: XChar.Character ← XChar.not] RETURNS [Number];
```

ReadNumberX is the same as **ReadNumber** except that the **numberParms** and **decimalSep** can be specified. If **decimalSep** is defaulted then the decimal separator described in the XComSoft messages file is used. See the XDigits chapter for more information. Fine Point: This procedure is currently exported through **XLRealX2**.

```
FormatReal: PROCEDURE [h: XFormat.Handle ← NIL, r: Number, width: NATURAL];
```

FormatReal formats **r** into a field **width** elements wide and passes the resulting text to **h**. If the number does not fit the field is filled with '> characters. **h.proc** may be called more than once.

```
PictureReal: PROCEDURE [
   h: XFormat.Handle ← NIL, r: Number, template: XString.Reader];
```

PictureReal is not implemented. Use **FormatNumber** for control over output.

```
NumberFormat: TYPE = RECORD [
   columns: ColumnCount ← 0,
   type: FormatType ← fixed,
   fill: Fill ← [none, 0],
   digitSpec: SELECT choice: DigitChoice FROM
      fractions = > [fractions: DigitCount],
      digits = > [digits: DigitCount],
      ENDCASE ← digits[0]];

ColumnCount: TYPE = [0..256);

FormatType: TYPE = {fixed, scientific, automatic};

Fill: TYPE = RECORD [type: FillType, nChars: ColumnCount];

FillType: TYPE = {zero, blank, none};

DigitCount: TYPE = [0..accuracy];

DigitChoice: TYPE = {fractions, digits};
```

NumberFormat is used by **FormatNumber** to allow precise control over the formatted number. **columns** specifies the number of columns required; zero means use as many as needed. **type** specifies whether the number is formated in fixed notation (**fixed**), scientific notation (**scientific**) or fixed if it will fit in the number of columns or scientific otherwise (**automatic**). **fill** specifies the type and number of characters of additional fill that is required before the number. **zero** and **blank** mean use zeros or blanks, respectively, as the fill characters, while **none** means use not fill. **digitSpec** allows the number of digits or fractions to be specified. **fractions[n]** means have n digits to the right of the decimal. **digits[n]** means have n significant digits in the number, except that **digits[0]** means have any number of digits in the number.

If **type = automatic** and **columns = 0**, then **f.digits** is used to determine whether the number will be fixed or scientific. This allows a client to say "I want **f.digits** of significant digits and you figure out whether it should be fixed or scientific." Using **type = automatic** and **columns#0** means that **FormatNumber** will use **columns** to determine whether the number is fixed or scientific, but often the client wants to specify significant digits rather than number of columns, especially since the number of columns will vary depending on the format and the value of the number.

```
FormatNumber: PROC [
  h: XFormat.Handle ← NIL, r: Number,
  format: NumberFormat ← defaultFormat, signalIfWontFit: BOOLEAN ← FALSE];

defaultFormat: NumberFormat = [
  columns: 0, type: fixed, fill: [none, 0], digitSpec: digits[0]];

WontFit: SIGNAL;
```

FormatNumber formats **r** according to **format**. If **format** specifies a number of columns, and the formatted number will not fit and **signalIfWontFit** is TRUE, the signal **WontFit** will be raised, If it is resumed or the number won't fit and **signalIfWontFit** is FALSE, **format.columns** '> will be passed to **h.proc**. If the number of columns or digit specification restricts the number of significant digits presented, the number of significant digits is reduced by rounding. If **h** is defaulted, output goes to the default output sink. (See the XFormat chapter for more details.) The default format specifies fixed notation in as many columns as necessary with no additional fill. If **r** is negative, the negative sign appears before zero fill and after blank fill. **h.proc** may be called more than once. Fine Point: This procedure is currently exported through **XLRealX**.

```
FormatNumberX: PROC [
  h: XFormat.Handle ← NIL, r: Number,
  format: NumberFormat ← defaultFormat, signalIfWontFit: BOOLEAN ← FALSE,
  numberParms: XDigits.NumberParms ← XDigits.defaultNumberParms,
  decimalSep: XChar.Character ← XChar.not];
```

FormatNumberX is the same as **FormatNumber** except that the **numberParms** and **decimalSep** can be specified. If **decimalSep** is defaulted then the decimal separator described in the XComSoft messages file is used. See the XDigits chapter for more details. Fine Point: This procedure is currently exported through **XLRealX2**.

65.2.4 Comparison

```
Comparison: TYPE = {less, equal, greater};
```

Compare: PROCEDURE [a, b: Number] RETURNS [Comparison];

Less, LessEq, Equal, GreaterEq, Greater, NotEq: PROCEDURE [
  a, b: Number] RETURNS [BOOLEAN];

Any of the compare operations may raise **Error[notANumber]**.

65.2.5 Operations

Add, Subtract, Multiply, Divide, Remainder: PROCEDURE [
  a, b: Number]RETURNS [Number];

Add, Multiply, Divide and **Remainder** may raise **Error[notANumber]** and **Error[overflow]** **Divide** may also raise **Error[divideByZero]**.

Exp: PROCEDURE [Number] RETURNS [Number];

Exp computes the results by continued fractions. **Exp** may raise **Error[underflow]**, **Error[notANumber]** and **Error[overflow]**.

Log: PROCEDURE [base, arg: Number] RETURNS [Number];

Log computes the logarithm to the base **base** of **arg** by **Ln(arg)/Ln(base)**. **Log** may raise **Error[overflow]**, **Error[invalidOperation]**, and **Error[notANumber]**.

Ln: PROCEDURE [Number] RETURNS [Number];

Ln may raise **Error[notANumber]** , **Error[overflow]** , and **Error[invalidOperation]**.

Power: PROCEDURE [base, exponent: Number] RETURNS [Number];

Power calculates **base** to the **exponent** power by e(**exponent*Ln(base)**). **Power** may raise **Error[notANumber]** and **Error[overflow]**.

Root: PROCEDURE [index, arg: Number] RETURNS [Number];-

Root calculates the **index** root of **arg** by e(**Ln(arg)/index**). Root may raise **Error[overflow]**, **Error[notANumber]**, and **Error[underflow]**.

SqRt: PROCEDURE [Number] RETURNS [Number];

SqRt calculates the square root of the input value by Newton's iteration. **SqRt** may raise **Error[notANumber]** and **Error[invalidOperation]**.

Abs, Negative, Double, Half: PROCEDURE [Number] RETURNS [Number];

Abs, Negative, Double and **Half** may raise **Error[notANumber]** **Double** may also raise **Error[overflow]**.

Cos: PROCEDURE [radians: Number] RETURNS [cos: Number];
Sin: PROCEDURE [radians: Number] RETURNS [sin: Number];
Tan: PROCEDURE [radians: Number] RETURNS [tan: Number];

Computes the trigonometric function by polynomial. Angles are measured in radians measured counterclockwise from the positive x axis about the origin [0, 0]. **Sin**, **Cos** and **Tan** may raise **Error[notANumber]** and **Error[invalidOperation]** (if **radians.exponent**>11). **Tan** may also raise **Error[overflow]**.

```
ArcCos PROCEDURE [x: Number] RETURNS [radians: Number];
ArcSin PROCEDURE [x: Number] RETURNS [radians: Number];
ArcTan PROCEDURE [x: Number] RETURNS [radians: Number];
```

Transcendental functions have an accuracy of about 1×10^{-11}. **ArcCos**, **ArcSin** and **ArcTan** may raise **Error[notANumber]**. **ArcSin** may also raise **Error[invalidOperation]** (if x NOT IN [-1..1]).

65.2.6 Special Numbers

A client can create special numbers that will cause the **Error[notANumber]** to be raised if used in any arithmetic operation.

```
SpecialIndex: TYPE = NATURAL;
```

```
MakeSpecial: PROCEDURE [index: SpecialIndex] RETURNS [Number];
IsSpecial: PROCEDURE [Number] RETURNS [yes: BOOLEAN, index: SpecialIndex];
```

65.2.7 Errors

```
Error: ERROR [code: ErrorCode];
```

```
ErrorCode: TYPE = {bug, divideByZero, invalidOperation, notANumber, overflow,
   underflow, unimplemented};
```

notANumber	means the number passed in is a special number.

65.2.8 Special Constants

```
zero: Number = LOOPHOLE[ Bits[0, 0, 0] ];
```

```
Pi: PROCEDURE RETURNS [Number];
E: PROCEDURE RETURNS [Number];
```

65.3 Usage/Examples

65.3.1 Special Numbers

```
--Make the special number

special:XLReal.Number ← XLReal.MakeSpecial[1];

--do some computations with Numbers

...
--If a problem occurs during computation, assign
```

```
n← special
...

[] ←XLReal.Ln[n];
--If n = special this call to XLReal.Ln will raise Error[notANumber]
```

65.3.2 Times of Common Operations

For the four arithmetic operations, typical timings (in microseconds) compared with the current Common Software 32-bit IEEE floating-point package (with no microcode assist) are:

	XLReal	REAL
Add or Subtract	500	800
Multiply	800	1000
Divide	1500	1900

65.4 Index of Interface Items

66

XMessage

66.1 Overview

The **XMessage** interface supports the multilingual requirements of systems requiring that the text to be displayed to the user be separable from the code and algorithms that utilize it. This allows workstation applications to define messages and developers and translators to supply the international representations of the text. The **XMessage** interface defines the message transfer mechanism necessary for applications to define application-specific messages, register them with the system, and access them.

The **XMessage** interface is part of the entire message machinery that provides multilingual text. Applications must be written to rely on messages for their text. A tool translates messages and produces a file containing the translated version of the messages.

66.1.1 Message Usage

Applications define collection of messages and refer to them by using a **Handle**. A unique key relative to that handle represents each message. To get the text of a message, the client calls **Get** or **GetList**. During development of applications, message handles are obtained by calling **AllocateMessages** and **RegisterMessages**. When the development is completed and a message file is generated, message handles are obtained by calling **MessagesFromFile** or **MessagesFromReference**.

Applications should be broken into three parts: the main code of the application that uses the messages, the code that defines and initializes the messages, and the code that gets message handles from the message file.

66.1.2 Message Composition and Templates

Frequently, text presented to the user should include items like names and sizes of objects, dates,and so forth. When defining such messages, it is best to define a single message template that allows certain fields to be filled in with this information. The piecemeal approach to constructing a understandable sentence normally does not work when the message is translated to a different language.

Templates are messages that will have additional text merged into them. The fields in templates are defined by numbers enclosed in angle brackets if the template contains multiple fields, or simply by angle brackets, if there is only one field.

66.2 Interface Items

66.2.1 Handles

```
Handle: TYPE = LONG POINTER TO Object;
```

```
Object: TYPE;
```

A **Handle** represents a collection of messages. It is normally associated with a particular application. It is obtained from the **AllocateMessages** operation and is a parameter of most operations.

66.2.2 Getting Messages

```
Get: PROCEDURE [h: Handle, msgKey: MsgKey] RETURNS [msg: XString.ReaderBody];
```

Get returns the message corresponding to the given message key within the group of messages specified by **h**.

```
GetList: PROCEDURE [h: Handle, msgKeys: MsgKeyList, msgs: StringArray];
```

```
MsgKeyList: TYPE = LONG DESCRIPTOR FOR ARRAY OF MsgKey;
```

```
StringArray: TYPE = LONG DESCRIPTOR FOR ARRAY OF XString.ReaderBody;
```

GetList fills the array of reader bodies with the bodies of the messages whose keys are in the message key list. This procedure is equivalent to:

```
FOR i IN [0..msgKeys.LENGTH) DO msgs[i] ← Get[msgKeys[i]]; ENDLOOP.
```

This procedure raises **Error[invalidMsgKeyList]** if **msgKeys** is NIL, **Error[invalidStringArray]** if **msgs** is NIL, and **Error[arrayMismatch]** if the lengths of the two descriptors are not equal.

66.2.3 Composing Messages

```
ComposeToFormatHandle: PROCEDURE [
   source: XString.Reader, destination: XFormat.Handle, args: StringArray];
```

```
Compose: PROCEDURE [
   source: XString.Reader, destination: XString.Writer, args: StringArray];
```

ComposeToFormatHandle and **Compose** compose a message by replacing the fields in **source** with the text in **args**. **ComposeToFormatHandle** uses an **XFormat.Handle** as the destination of the message, while **Compose** uses an **XString.Writer**. A field is specified by a number enclosed in angle brackets. These operations may raise **Error[invalidString]** if **source** is empty and **Error[notEnoughArguments]** if **args** is NIL. To maintain backward compatibility with existing messages, the string array is one origin (that is, the field <1> accesses **args[0]**;

```
ComposeOneToFormatHandle: PROCEDURE [
   source: XString.Reader, destination: XFormat.Handle, arg: XString.Reader];
```

```
ComposeOne: PROCEDURE [
   source: XString.Reader, destination: XString.Writer, arg: XString.Reader];
```

ComposeOneToFormatHandle and **ComposeOne** compose a message by replacing the single field in **source** with **arg**. **ComposeOneToFormatHandle** uses an **XFormat.Handle** as the destination of the message, while **ComposeOne** uses an **XString.Writer**. The single field is specified by empty angle brackets, < >. These operations may raise **Error[invalidString]** if **source** is empty and **Error[notEnoughArguments]** if **arg** is NIL.

```
Decompose: PROCEDURE [source: XString.Reader] RETURNS [args: StringArray];
```

Decompose currently does nothing.

66.2.4 Defining Messages

Messages are defined by constructing an array of message entries and registering them with the system.

```
Messages: TYPE = LONG DESCRIPTOR FOR ARRAY OF MsgEntry;
```

```
MsgEntry: TYPE = RECORD [
   msgKey: MsgKey,
   msg: XString.ReaderBody,
   translationNote: LONG STRING ← NIL,
   translatable: BOOLEAN ← TRUE,
   type: MsgType ← userMsg,
   id: MsgID];
```

```
MsgKey: TYPE = CARDINAL;
```

```
MsgType: TYPE = {userMsg, template, argList, menuItem, pSheetItem, commandItem,
   errorMsg, infoMsg, promptItem, windowMenuCommand, others};
```

```
MsgID: TYPE = CARDINAL;
```

Messages describes a group of message entries and is a parameter to **RegisterMessages**. A **MsgEntry** contains information about each message. The **msgKey** field is the **Handle**-relative key of the message. The **msg** field contains the text of the message itself, while all other fields are to help in the translation process. The **tranlationNote** field provides notes to the translator. The **translatable** boolean indicates whether the message should be translated. The **MsgType** enumerated provides a hint of how the message will be used. The **MsgID** is a unique identifier for the message. For a given group of messages, each message should have a unique value for its **MsgID**. The **MsgID** must remain unique for all time, across all releases. This ID allows the translators to determine when a new message has been added or an old message deleted.

```
AllocateMessages: PROCEDURE [
   applicationName: LONG STRING, maxMsgIndex: CARDINAL,
   clientData: ClientData, proc: DestroyMsgsProc]
   RETURNS [h: Handle];

ClientData: TYPE = LONG POINTER;

DestroyMsgsProc: TYPE = PROCEDURE [clientData: ClientData];
```

AllocateMessages allows a client to define a domain of messages for subsequent registry and access. All access to messages will be relative to the returned handle. The **applicationName** parameter names the message domain to the message implementation. **maxMsgIndex** defines the maximum number of messages that are registered for this domain. The **ClientData** and **DestroyMsgsProc** parameters are provided to notify the client when the **DestroyMessages** operation is invoked.

```
RegisterMessages: PROCEDURE [
   h: Handle, messages: Messages, stringBodiesAreReal: BOOLEAN];
```

RegisterMessages allows a client to initialize a domain of messages. It uses the **stringBodiesAreReal** boolean to decide whether to copy the byte sequences of the messages. If **stringBodiesAreReal** is FALSE, it copies the reader body and bytes of the messages field in each entry of **messages**. If it is TRUE, **RegisterMessages** copies the reader body of the entry and relies on the bytes to not be deallocated until after a call to **DestroyMessages**.

66.2.6 Obtaining Messages from a File

```
MessagesFromFile: PROCEDURE [
   fileName: LONG STRING, clientData: ClientData, proc: DestroyMsgsProc]
   RETURNS [msgDomains: MsgDomains];

MessagesFromReference: PROCEDURE [
   file: NSFile.Reference, clientData: ClientData, proc: DestroyMsgsProc]
   RETURNS [msgDomains: MsgDomains];

MsgDomains: TYPE = LONG DESCRIPTOR FOR ARRAY OF MsgDomain;

MsgDomain: TYPE = RECORD [
   applicationName: XString.ReaderBody,
   handle: Handle];
```

MessagesFromFile and **MessagesFromReference** return a sequence of message domains that are name, message handle pairs. **MessagesFromFile** gets the messages from the file named **fileName** in the system folder, while **MessagesFromReference** gets the messages from the file whose reference is **file**. Storage for **msgDomains** must by freed by calling **FreeMsgDomainStorage**. The **ClientData** and **DestroyMsgsProc** parameters are provided to notify the application when the **DestroyMessages** operation is invoked.

```
FreeMsgDomainsStorage: PROCEDURE [msgDomains: MsgDomains];
```

66.2.7 Destroying Message Handles

```
DestroyMessages: PROCEDURE [h: Handle];
```

DestroyMessages invokes the **DestroyMsgsProc** associated with the handle and then frees any resources that are currently associated with **h**. The handle should no longer be used.

66.2.6 Error

```
Error: ERROR [type: ErrorType];

ErrorType: TYPE = {
  arrayMismatch, invalidArgIndex, invalidMsgKey, invalidMsgKeyList,
  invalidStringArray, invalidString, notEnoughArguments};
```

66.3 Usage/Examples

66.3.1 Structuring Applications to Use Messages

Applications that use messages have at least two parts. The first part is the code for the application's functions. It is produced by programming tools such as the compiler and binder with Mesa source programs as input. The second part consists of the messages that provide text to the user. The application defines its messages and provides initial information to the translators of the messages.

Messages that are to be translated must be able to communicate in a precise type-safe way. Throughout the translation process, it should be possible to verify the message with its original version.

The cleanest and safest possible interface between application developers (message definers) and translators is to deliver a bcd that contains all the messages used by the application as well as a well-defined mechanism for communicating them to some client. The **RegisterMessages** procedure provides the mechanism; all else that is needed is to avoid other distractions (like importing or exporting of application private facilities). To that end, the following conventions are proposed for modules/configurations that define and register messages:

1. Isolate message definition code into modules whose sole function is to define and register the message text for the application.

2. Allocate the **Handle** and register all messages via the modules' configuration's START code.

3. If multiple modules are required to define the application messages, provide a configuration that starts all modules in the the correct order and provides the correct IMPORTS and EXPORTS.

4. **XMessage** definition modules and configurations must not depend upon application-specific facilities. The IMPORTS list of any message-defining module should be restricted

to procedures defined in the **XMessage** interface (such as **RegisterMessages**, **AllocateMessages**,and so forth).

As a consequence of 2 and 4 above, applications must provide a mechanism for communicating the **Handle** between suppliers of messages (callers of **ResisterMessages**) and users of messages (callers of **GetMsg**, and and so forth) . A simple solution is to have the message-definition module export a procedure that returns the handle.

66.3.2 Example of Message Usage

The following message example has three segments. The first is an interface that defines the messages for the example. The second is the module that provides the raw material for the messages. This module is used to supply the message text while running the application while it is being developed. It is used to supply the raw data to the message translators. The third part is the module that uses the messages.

```
-- ExampleMessage.mesa

DIRECTORY
   XMessage USING [Handle];

ExampleMessage: DEFINITIONS = BEGIN

   Keys: TYPE = MACHINE DEPENDENT {
      delete(0), confirmDelete(1), deleteDone(3)...};

   GetHandle: PROCEDURE RETURNS[h: XMessage.Handle];

   END. -- of ExampleMessage:


-- ExampleMessageImpl.mesa

DIRECTORY ...

ExampleMessageImpl: PROGRAM
   IMPORTS ...
   EXPORTS ExampleMessage = BEGIN

   h: XMessage.Handle;

   GetHandle: PUBLIC PROCEDURE RETURNS [XMessage.Handle] = {RETURN[h]};

   DeleteMessages: XMessage.DestroyMsgsProc = {};

   Init: PROCEDURE = {
      msgArray: ARRAY Keys OF XMessage.MsgEntry ← [
         delete: [
            msgKey: Keys.delete.ORD,
            msg: XString.FromSTRING["Delete"L],
            translationNote: "Delete command name"L,
            translatable: TRUE,
            type: menuItem,
            id: 0],
        confirmDelete: [
          msgKey: Keys.confirmDelete.ORD,
          msg: XString.FromSTRING["Are you sure you want to delete that item"L],
          translatable: TRUE,
          type: userMsg,
          id: 1],
        deleteDone: [
          msgKey: Keys.deleteDone.ORD,
          msg: XString.FromSTRING["The item <1> has been deleted"L],
          translatable: TRUE,
          type: template,
          id: 3]];

      h ← XMessage.AllocateMessages[
        "Example"L, Keys.LAST.ORD.SUCC, NIL, DeleteMessages];
      XMessage.RegisterMessages[h, LOOPHOLE[LONG[DESCRIPTOR[msgArray]]], FALSE]};

    Init[];

    END. -- of ExampleMessageImpl

  -- ExampleImpl.mesa

  DIRECTORY ...

  ExampleImpl: PROGRAM
    IMPORTS XMessage, ExampleMessage = BEGIN

    h: XMessage.Handle = ExampleMessage.GetHandle[]

    DeleteOne: PROCEDURE [ . . . ] = {
      r: XString.Reader = XMessage.Get[h, ExampleMessage.Keys.confirmDelete.ORD];
      . . .
      };

    }. -- of ExampleImpl
```

66.4 Interface Item Index

67

XString

67.1 Overview

The **XString** interface is part of a string package that supports the Xerox Character Standard. It provides the basic data structures for representing encoded sequences of characters and some operations on these data structures.

67.1.1 Character Standard

The *Xerox Character Code Standard* defines a large number of characters, encompassing not only familiar ASCII characters but Japanese and Chinese Kanji characters and other characters to provide a comprehensive character set able to handle international information-processing requirements. Because of the large number of characters, the data structures in **XString** are more complicated than a **LONG STRING**'s simple array of ASCII characters, but the operations provided are more comprehensive.

Characters are 16-bit quantities that are composed of two 8-bit quantities: their character set and character code within a character set. The Character Standard defines how characters may be encoded, either as runs of 8-bit character codes of the character set or as 16-bit characters where the character set and character code are in consecutive bytes. See the XChar chapter for information and operations on characters.

67.1.2 Data Structures

Three main data structures are defined by **XString**: **Context**, **ReaderBody** and **WriterBody**. Contexts provide information for determining how characters are encoded. Reader bodies and readers describe a sequence of readonly characters. Writer bodies and writers describe a sequence of writeable characters.

A **Context** contains information about how characters are encoded in the byte sequence. The **suffixSize** field describes whether the first byte is encoded as an 8-bit character or a 16-bit character, the **prefix** field contains the character set of the first character if the encoding is 8-bit characters, and the **homogeneous** field is **TRUE** only if there are no character set shifts in the sequence of characters.

A **ReaderBody** describes some readonly characters that are stored as a sequence of bytes. The reader body contains a pointer to the allocation unit containing the bytes, **bytes**, the offset from the pointer to the first byte, **offset**, the offset from the pointer of the byte after the last byte in the byte sequence, **limit**, and the context information describing how the first character is encoded, **context**. Most clients should not have to access fields of a reader body. Many operations take a **Reader**, a pointer to a reader body, as a parameter to reduce the number of words of parameters.

A **WriterBody** describes some characters that may be edited. In addition to containing all the information in a reader body, it also contains an offset from the pointer to the first character not in the allocation unit, **maxLimit**, the context that describes how the last character is encoded, **endContext**, and the zone that contains the allocation unit, **zone**. Writer bodies are typically passed by reference.

The designers of **XString** felt there is a fundamental difference between a string that will only be read and one that will be constructed. They felt that the major usage of strings was to describe and examine existing strings, not construct new ones. This difference is reflected in the two types, readers and writers.

67.1.3 Operations

There are a wide range of operations on both readers and writers. Some operations that return simple information about readers, such as **ByteLength** and **Empty**. Others access characters in a reader, such as **First**, **NthCharacter**, and **Lop**. The operation **ReaderFromWriter** can be used to convert a writer to a reader.

There are operations that create reader bodies from other data structures, such as **FromSTRING**. Similarily, there are operations that create writer bodies from other structures, such as **WriterBodyFromSTRING**.

Routines allocate and deallocate byte sequences of both readers and writers. **CopyToNewReaderBody** makes a copy of the characters of a reader. **NewWriterBody** will create an empty writer body that can hold a given number of bytes. **CopyToNewWriterBody** is similar to **NewWriterBody** but initializes the writer with a given reader.

Other operations compare the characters in readers: **Equal** checks for equality and **Compare** does a multinational lexical comparison. There are operations for scanning readers for specific characters. The operation **ReaderToNumber** converts a reader to a numeric value. There are **Courier** description routines for both readers and reader bodies. There is also support for backward-accessing characters in a reader.

Routines are provided for appending to writers and editing writers. **AppendReader** appends the characters of a reader to a writer. **ReplacePiece** provides a general editing operation for writers. Only the basic appending primitives are provided in **XString**. The **XFormat** interface can be used to append converted values, such as numbers, to writers.

67.2 Interface Items

67.2.1 Contexts

```
Context: TYPE = MACHINE DEPENDENT RECORD [
   suffixSize(0:0..6): [1..2],
   homogeneous(0:7..7): BOOLEAN,
   prefix(0:8..15): Byte];
```

A **Context** contains information about how characters are encoded in the byte sequence. The **suffixSize** field describes whether the first byte is encoded as an 8-bit character or a 16-bit character, the **prefix** field contains the character set of the first character if the encoding is 8-bit characters, and the **homogeneous** field is TRUE only if there are no character set shifts in the sequence of characters.

The Character Set Standard describes how characters may be encoded as a sequence of bytes. They call the 8-bit character encoding stringlet8 and call the 16-bit character encoding stringlet16. In 8-bit character encoding, consecutive bytes contain character codes of characters in the same character set. In 16-bit character encoding, the character set and character code are contained in consecutive bytes. The **suffixSize** field describes how the characters are encoded; it is 1 for 8-bit character encoding and 2 for 16-bit character encoding.

The **prefix** field contains the character set of the first character if it is an 8-bit encoded character. Subsequent characters in the string use this same prefix unless a character set or encoding transition is encountered. It is not used for 16-bit encoded characters.

The **homogeneous** field is an accelerator. If it is TRUE, some operations may be faster. It is important to set it TRUE only if the byte sequence contains no character set shifts. It is always safe to set it to FALSE.

```
emptyContext: Context = [suffixSize: 1, homogeneous: TRUE, prefix: 0];

vanillaContext: Context = [suffixSize: 1, homogeneous: FALSE, prefix: 0];

unknownContext: Context = [suffixSize: 1, homogeneous: FALSE, prefix: 377B];
```

emptyContext, **vanillaContext**, and **unknownContext** are three **Context** constants. An empty writer should have **emptyContext** as its context and endContext. **vanillaContext** is the default context. **unknownContext** signifies that the context is unknown. It is generally used only for an end context, a context that describes the last character of a sequence.

67.2.2 Readers and ReaderBodies

```
Reader: TYPE = LONG POINTER TO ReaderBody;

ReaderBody: TYPE = PRIVATE MACHINE DEPENDENT RECORD [
   context(0): Context,
   limit(1): CARDINAL,
   offset(2): CARDINAL,
   bytes(3): ReadOnlyBytes];

ReadOnlyBytes: TYPE = LONG POINTER TO READONLY ByteSequence;

ByteSequence: TYPE = RECORD [
   PACKED SEQUENCE COMPUTED CARDINAL OF Byte];
```

```
Byte: TYPE = Environment.Byte;
```

A **ReaderBody** describes some readonly characters that are stored as a sequence of bytes. The reader body contains a pointer to the allocation unit containing the bytes, **bytes**, the offset from the pointer to the first byte, **offset**, the offset from the pointer of the byte after the last byte in the byte sequence, **limit**, and the context information describing how the first character is encoded, **context**. Most clients should not have to access fields of a reader body. Reader bodies can be thought of as fat pointers. Many operations take a **Reader**, a pointer to a reader body, as a parameter to reduce the number of words of parameters.

A **ReaderBody** is like a substring descriptor. The **offset** and **limit** fields can be changed to describe a subsequence of bytes. Routines such as **Lop** and **ScanForCharacter** take advantage of this substring-like behavior.

```
nullReaderBody: ReaderBody = [
   limit: 0, offset: 0, bytes: NIL, context: vanillaContext];
```

nullReaderBody defines a null value for a reader body. To test for an empty reader body, it should *not* be compared to **nullReaderBody**. The operation **Empty** should be used instead.

67.2.3 Writers and WriterBodies

```
Writer: TYPE = LONG POINTER TO WriterBody;

WriterBody: TYPE = PRIVATE MACHINE DEPENDENT RECORD [
   context(0): Context,
   limit(1): CARDINAL,
   offset(2): CARDINAL,
   bytes(3): Bytes,
   maxLimit(5): CARDINAL,
   endContext(6): Context,
   zone(7): UNCOUNTED ZONE];

Bytes: TYPE = LONG POINTER TO ByteSequence;
```

Writers describe a sequence of bytes that may be changed. Writers are bit-wise compatible with a reader and contain additional information for storage management and appending characters. The **maxLimit** field describes the limits of the allocation unit, the **zone** field is the zone used for allocating and freeing the bytes, and the **endContext** field is an accelerator for operations that append characters.

By including a zone in the writer body, operations that add characters to the writer are able to allocate a larger byte sequence, copy the old bytes, and update the byte pointer in the writer body without invalidating the writer variable that the caller owns.

```
nullWriterBody: WriterBody = [
   limit: 0, offset: 0, bytes: NIL, context: vanillaContext, maxLimit: 0,
   endContext: vanillaContext, zone: NIL];
```

nullWriterBody defines a null value for a writer body.

67.2.4 Simple Reader Operations

ByteLength: PROCEDURE [r: Reader] RETURNS [CARDINAL] = INLINE ...;

ByteLength returns the number of bytes in r. If r is NIL, it returns zero.

CharacterLength: PROCEDURE [r: Reader] RETURNS [CARDINAL];

CharacterLength returns the number of logical characters in r. Floating accent characters are treated as separate logical characters. If r is NIL it returns zero. If r is a valid reader, then ByteLength[r] = 0 iff CharacterLength[r] = 0. If r contains an invalid encoding, CharacterLength will raise the error InvalidEncoding.

Dereference: PROCEDURE [r: Reader] RETURNS [rb: ReaderBody];

Dereference returns nullReaderBody if r is NIL and r↑ otherwise.

Empty: PROCEDURE [r: Reader] RETURNS [BOOLEAN] = INLINE ...;

Empty returns TRUE if r is NIL or ByteLength[r] = 0.

ReaderInfo: PROCEDURE [r: Reader] RETURNS [context: Context, startsWith377B: BOOLEAN] ;

ReaderInfo returns the context of the reader and whether the first byte of the reader is 377B, the character set shift code.

67.2.5 Accessing Characters

Because of the large number of characters in the character set standard and the way they are encoded, it is normally not possible to access characters of a reader by indexing. Instead, a number of operations are provided to access characters.

Character: TYPE = XChar.Character;

First: PROCEDURE [r: Reader] RETURNS [c: Character];

First returns the first logical character. It is equivalent to NthCharacter[s, 0] but is usually more efficient. If Empty[r] then XChar.not is returned. It may raise the InvalidEncoding error.

NthCharacter: PROCEDURE [r: Reader, n: CARDINAL] RETURNS [c: Character];

NthCharacter returns the nth logical character. Floating accent characters are treated as separate logical characters. First should be used if n = 0. If CharacterLength[r] <= n then XChar.not is returned. It may raise the InvalidEncoding error.

Lop: PROCEDURE [r: Reader] RETURNS [c: Character];

Lop removes the first character from the front of a reader and returns it. If r is empty it returns XChar.not. If r contains one logical character, Lop sets r to be empty and returns the first logical character. Otherwise, **Lop** modifies **r** to point to the second logical character and returns the first. It may raise the **InvalidEncoding** error.

Map: PROCEDURE [r: Reader, proc: MapCharProc] RETURNS [c: Character];

MapCharProc: TYPE = PROC [c: Character] RETURNS [stop: BOOLEAN];

Map enumerates the reader, calling **proc** once for each character in **r**. If **proc** returns **TRUE** it returns that character; otherwise it returns **XChar.not**. It is equivalent to:

```
FOR i: CARDINAL IN [0..CharacterLength[r]) DO
  IF proc[c ← NthCharacter[r, i]] THEN RETURN[c]; ENDLOOP
RETURN[XChar.not] ;
```

Map may raise the **InvalidEncoding** error.

67.2.6 Errors

Error: ERROR [code: ErrorCode] ;

```
ErrorCode: TYPE = {
  invalidOperation, multipleCharSets, tooManyBytes, invalidParameter};
```

invalidOperation	an invalid operation has been attempted.
multipleCharSets	**InitBreakTable** has been called with a reader that contains multiple character sets.
tooManyBytes	a **LONG STRING** has been passed to **FromSTRING** or **WriterBodyFromSTRING** and the string contains too many bytes. These operations use the string as the byte pointer so the offset is non-zero, reducing the number of bytes it may hold. This is also raised by **CopyReader** for a similar reason.
invalidParameter	The term **invalidParameter** means an operation has been invoked with an invalid parameter.

InvalidEncoding: ERROR [invalidReader: Reader, firstBadByteOffset: CARDINAL] ;

The error **InvalidEncoding** is raised by operations when they detect a sequence of bytes that is not a valid character encoding. While two character set shifts with no intervening character is an invalid encoding according to the character standard, only **ValidateReader** will raise **InvalidEncoding** if it detects that case. The other operations will ignore the first character set shift. Invalid encodings include ending with a character set shift or partial character set shift and having a non-zero byte following two 377B bytes.

67.2.7 Conversion to Readers

ReaderFromWriter: PROCEDURE [w: Writer] RETURNS [Reader] = INLINE ... ;

ReaderFromWriter provides a conversion from the type **Writer** to the type **Reader**. This operation takes advantage of the fact that the first part of writer bodies are bit-wise compatible with reader bodies, and hence this operation simply loopholes the writer into the reader.

FromBlock: PROCEDURE [
  block: Environment.Block, context: Context ← vanillaContext] RETURNS [ReaderBody];

FromBlock returns a reader body that describes the block.

FromChar: PROCEDURE [char: LONG POINTER TO Character] RETURNS [ReaderBody];

FromChar returns a reader body that describes the character. The pointer to the character must remain valid for the lifetime of the reader body.

FromNSString: PROCEDURE [s: NSString.String, homogeneous: BOOLEAN ← FALSE]
  RETURNS [ReaderBody];

FromNSString returns a reader body that describes the characters in the NSString. The context of the reader body will be [suffixSize: 1, homogeneous: homogeneous, prefix: 0].

FromSTRING: PROCEDURE [s: LONG STRING, homogeneous: BOOLEAN ← FALSE]
  RETURNS [ReaderBody];

FromSTRING returns a reader body that describes the characters in the LONG STRING. The context of the reader body will be [suffixSize: 1, homogeneous: homogeneous, prefix: 0]. This operation may raise Error[tooManyBytes] if the string contains more than CARDINAL.LAST - StringBody.SIZE * Environment.bytesPerWord bytes.

67.2.8 Reader Allocation

CopyReader: PROCEDURE [r: Reader, z: UNCOUNTED ZONE] RETURNS [new: Reader];

CopyReader makes a copy of the reader body and characters of r, allocating from z as a single allocation unit, the byte sequence for the characters, and the reader body. Note that this operation returns a reader while all other operations in this interface that create a reader or reader body return the reader body. The reason is to avoid a double allocation problem in which the byte sequence and reader body are allocated from two separate nodes. FreeReaderBytes can be used to free the new reader and the associated bytes. Note: This operation may raise Error[tooManyBytes] if the reader contains more than CARDINAL.LAST - ReaderBody.SIZE * Environment.bytesPerWord bytes. Errors in allocating from the zone are allowed to propagate.

CopyToNewReaderBody: PROCEDURE [r: Reader, z: UNCOUNTED ZONE] RETURNS [ReaderBody];

CopyToNewReaderBody allocates a copy of the bytes of r using z and returns a reader body describing them. If r is NIL it returns nullReaderBody. Errors in allocating from the zone are allowed to propagate.

FreeReaderBytes: PROCEDURE [r: Reader, z: UNCOUNTED ZONE];

FreeReaderBytes may be used to free the storage allocated by CopyReader and CopyToNewReaderBody. If r is non-NIL and z is non-NIL, it frees r.bytes to the zone. When the reader has been obtained from CopyReader, FreeReaderBytes will free the single allocation unit that contains both the reader body and byte sequence. When the reader has been obtained from **CopyToNewReaderBody**, **FreeReaderBytes** will free the allocation unit that contains the byte sequence but will not free the reader body. Errors in freeing to the zone are allowed to propagate.

67.2.9 Simple Writer Operations

ClearWriter: PROCEDURE [w: Writer];

ClearWriter makes **w** empty. It is analogous to the LONG STRING statement **s.length** ← 0.

WriterInfo: PROCEDURE [w: Writer]
  RETURNS [unused: CARDINAL, endContext: Context, zone: UNCOUNTED ZONE];

WriterInfo returns the number of allocated but unused bytes of a writer as well as its endContext and its zone.

67.2.10 Conversion to Writers

WriterBodyFromBlock: PROCEDURE [block: Environment.Block, inUse: CARDINAL ← 0]
  RETURNS [WriterBody];

WriterBodyFromBlock returns a writer body that describes the block. The writer body's offset and **maxLimit** fields are set from the block's **startIndex** and **stopIndexPlusOne** fields, respectively. The **inUse** parameter is used to set the limit field of the writer body. If the block's pointer is NIL or **inUse** is larger than the number of bytes in the block, **Error[invalidParameter]** is raised.

WriterBodyFromNSString: PROCEDURE [
  s: NSString.String, homogeneous: BOOLEAN ← FALSE] RETURNS [WriterBody];

WriterBodyFromNSString returns a writer body that describes the characters in the **NSString**. Its context is **[suffixSize: 1, homogeneous: homogeneous, prefix: 0]**.

WriterBodyFromSTRING: PROCEDURE [
  s: LONG STRING, homogeneous: BOOLEAN ← FALSE] RETURNS [WriterBody];

WriterBodyFromSTRING returns a writer body that describes the characters in the LONG STRING. Its context is **[suffixSize: 1, homogeneous: homogeneous, prefix: 0]**. This operation may raise **Error[tooManyBytes]** if the string contains more than CARDINAL.LAST - StringBody.SIZE * Environment.bytesPerWord bytes.

67.2.11 Writer Allocation

NewWriterBody: PROCEDURE [maxLength: CARDINAL, z: UNCOUNTED ZONE]
  RETURNS [WriterBody];

NewWriterBody allocates a byte sequence that has room for **maxLength** bytes using **z** and returns an empty writer body that contains the bytes. Errors in allocating **ByteSequence[maxLength]** from the zone are allowed to propagate.

```
CopyToNewWriterBody: PROCEDURE [
   r: Reader, z: UNCOUNTED ZONE, endContext: Context ← unknownContext,
   extra: CARDINAL ← 0] RETURNS [w: WriterBody];
```

CopyToNewWriterBody allocates a byte sequence that has room for **ByteLength[r] + extra** bytes using **z**, copies the bytes of **r** into the newly allocated byte sequence, and returns a writer body that contains the bytes. The end context of the writer body is **endContext**. Errors in allocating from the zone are allowed to propagate.

```
ExpandWriter: PROCEDURE [w: Writer, extra: CARDINAL];
```

ExpandWriter assures that at least **extra** bytes are available in the writer's bytes. If **w.zone** is NIL, then **Error[invalidOperation]** is raised. Errors in allocating a new byte sequence if required are allowed to propagate.

```
FreeWriterBytes: PROCEDURE [w: Writer];
```

FreeWriterBytes may be used to free the byte sequence of a writer as long as it was allocated from the writer's zone. It may be used to free the byte sequence of writers created by **CopyToNewWriterBody** and **NewWriterBody**. If **w** is non-NIL and **w.zone** is non-NIL, it frees **w.bytes** to the zone. Note that it does not free the writer body. Errors in freeing to the writer's zone are allowed to propagate.

67.2.12 Comparison of Readers

```
Equal: PROCEDURE [r1, r2: Reader] RETURNS [BOOLEAN];
```

Equal returns TRUE if and only if the number of logical characters is equal and the strings match when compared character by character, i.e., effectively **CharacterLength[r1] = CharacterLength[r2]** and **NthCharacter[r1, i] = NthCharacter[r2, i]** for i in the range **[0.. CharacterLength[r1])**. It may raise the **InvalidEncoding** error.

```
Equivalent: PROCEDURE [r1, r2: Reader] RETURNS [BOOLEAN];
```

Equivalent returns TRUE if and only if the number of logical characters is equal and the strings match when compared character by character, ignoring case. It is equivalent to:

```
IF CharacterLength[r1] # CharacterLength[r2] THEN RETURN[FALSE];
FOR i: CARDINAL IN [0..CharacterLength[r1]) DO
   IF Decase[NthCharacter[r1, i]] # Decase[NthCharacter[r2, i]] THEN RETURN[FALSE];
   ENDLOOP;
RETURN[TRUE].
```

Equivalent may raise the **InvalidEncoding** error.

```
SortOrder: TYPE = MACHINE DEPENDENT {
   standard(0), spanish(1), swedish(2), danish(3), firstFree(4), null(377B)};
```

SortOrder is a parameter to **Compare** and **CompareStringsAndStems** that specifies the sort order. **danish**, **spanish**, and **swedish** differ from **standard** only in some characters in character set zero.

```
Compare: PROCEDURE [
   r1, r2: Reader, ignoreCase: BOOLEAN ←TRUE, sortOrder: SortOrder ← standard]
   RETURNS [Relation];

CompareStringsAndStems: PROCEDURE [
   r1, r2: Reader, ignoreCase: BOOLEAN ←TRUE, sortOrder: SortOrder ← standard]
   RETURNS [relation: Relation, equalStems: BOOLEAN];

Relation: TYPE = {less, equal, greater};
```

Compare and **CompareStringsAndStems** compare two readers. They return a relation indicating the sorted relationship of their arguments with the case of characters optionally ignored during the comparison. In **CompareStringsAndStems**, **equalStems** will be TRUE if both readers are equal up to the length of the shorter. They may raise the **InvalidEncoding** error.

67.2.13 Numeric Conversion of Readers

```
ReaderToNumber: PROCEDURE [
   r: Reader, radix: CARDINAL ← 10, signed: BOOLEAN ←FALSE] RETURNS [LONG INTEGER];
```

ReaderToNumber converts the characters in the reader to a number. If **radix** is other than 8, 10, or 16, **XString.Error[invalidOperation]** is raised. The syntax for a number is:

{'- | '+} {*baseNumber*} {'b | 'B | 'd | 'D | 'h | 'H} {*scaleFactor*}

where {} indicates an optional part and "|" indicates a choice, and *baseNumber* and *scaleFactor* are sequences of digits. The value returned is ± *baseNumber* * *radix* ** *scaleFactor*. The *radix* depends on the contexts of **r** and **radix**. If **r** ends with a 'B or 'b, *radix* is 8; if it ends with a 'D or 'd, *radix* is 10, if it ends with a 'h or 'H, *radix* is 16; otherwise it is **radix**. The number *scaleFactor* is always expressed in radix 10. If **r** does not have valid form, or **r** does not contain any characters, or *radix* is 8 and non-octal digits are used, or **signed** is FALSE and the reader contains a minus sign, the signal **InvalidNumber** is raised. If it is resumed, the operation returns zero. If **signed** is FALSE and the number would overflow 2^{32}-1 or **signed** is TRUE and the number is not in the range [-2^{31} .. 2^{31}), the signal **Overflow** is raised. If it is resumed, the operation returns zero. **ReaderToNumber** may raise the **InvalidEncoding** error.

```
ReaderToNumberX: PROCEDURE [
   r: Reader, radix: CARDINAL ← 10, signed: BOOLEAN ←FALSE,
   numberParms: XDigits.NumberParms ← XDigits.defaultNumberParms]
   RETURNS [LONG INTEGER];
```

ReaderToNumberX is the same as **ReaderToNumber** except that the **numberParms** parameter is added. If **radix** is other than 10, **numberParms.representation** and **numberParms.thousandsSep** will be ignored. See the XDigits chapter for more details. Fine Point: This procedure is currently exported through **XStringX2**.

```
InvalidNumber: SIGNAL;
```

The signal **InvalidNumber** is raised by the string to number operations when the string is the wrong syntax for a number. Resuming this signal results in the operation returning zero.

```
Overflow: SIGNAL;
```

The signal **Overflow** is raised by the string to number operations when the string describes a number that is too large. Resuming this signal results in the operation returning zero.

67.2.14 Character Scanning

```
ScanForCharacter: PROCEDURE [r: Reader, char: Character, option: BreakCharOption]
  RETURNS [breakChar: Character, front: ReaderBody]

BreakCharOption: TYPE = {ignore, appendToFront, leaveOnRest};
```

ScanForCharacter scans the string for the first instance of **char**. If **char** is found in **r**, the characters before it will be described by **front** and the characters after it will be described by **r**. **char** will be on the end of **front**, discarded, or left on the front of **r↑** if **option** is **appendToFront**, **ignore**, or **leaveOnRest**, respectively. **char** will be returned as **breakChar**. If it does not encounter **char** in **r**, then **front** will be equal to **r↑** as it was when the procedure was invoked and **r↑** will be updated to be 0 characters long. **XChar.not** will be returned as **breakChar**. **ScanForCharacter** may raise the **InvalidEncoding** error.

```
Scan: PROCEDURE [r: Reader, break: BreakTable, option: BreakCharOption]
  RETURNS [breakChar: Character, front: ReaderBody];

BreakTable: TYPE = LONG POINTER TO BreakTableObject;

BreakTableObject: TYPE = RECORD
  otherSets: StopOrNot ← stop,
  set:Environment.Byte ← 0,
  codes: PACKED ARRAY [0..256) OF StopOrNot ← ALL[not]];

StopOrNot: TYPE = {stop, not} ← not;
```

Scan is like **ScanForCharacter** except that it can scan for any number of character codes in a particular character set. The **BreakTable** defines which character codes of a character set are scanned for. Scanning is searching for the first character, **c**, such that

```
    (XChar.Set[c] = break.set AND break.codes[XChar.Code[c]] = stop) OR
    (XChar.Set[c] # break.set AND break.otherSets = stop).
```

The disposition of the character that terminates scanning depends on **option** as in **ScanForCharacter**. If the character terminated scanning because it was in a different character set, **XChar.not** will be returned as **breakChar**. **Scan** may raise the **InvalidEncoding** error.

```
InitBreakTable: PROCEDURE [
  r: Reader, stopOrNot: StopOrNot, otherSets: StopOrNot]
  RETURNS [break: BreakTableObject];
```

InitBreakTable initializes a **BreakTableObject** to stop (or not stop) on the characters of r depending on **stopOrNot**. If r has multiple character sets, **Error[multipleCharSets]** is raised. **InitBreakTable** may raise the **InvalidEncoding** error.

67.2.15 Other Reader Operations

ComputeEndContext: PROCEDURE [r: Reader] RETURNS [c: Context];

ComputeEndContext returns the context of the last character in r. If **CharacterLength[r] = 0** then **emptyContext** is returned. **ComputeEndContext** may raise the **InvalidEncoding** error.

DescribeReader: Courier.Description;

DescribeReader is a Courier description routine. It is provided for clients that need to serialize and deserialize readers.

DescribeReaderBody: Courier.Description;

DescribeReaderBody is a Courier description routine. It is provided for clients that need to serialize and deserialize readers bodies.

Run: PROCEDURE [r: Reader] RETURNS [run: ReaderBody];

Run is like **Lop** except that it deals in homogeneous runs of characters instead of single characters. It will return a reader body describing the first homogeneous run of r and update r to remove the run. If **Empty[r]** it returns **nullReaderBody**. It may raise the **InvalidEncoding** error.

ValidateReader: PROCEDURE [r: Reader];

ValidateReader checks the bytes of r to ensure that it is a valid encoding. If it is not a valid encoding, the error **InvalidEncoding** is raised. Possible invalid encodings include ending in a character set shift with no character or following two successive 377B bytes by a non-zero byte. null run, two character set shifts with no intervening character, is an invalid encoding that is checked by **ValidateReader**. If the offset is greater than the limit or the byte pointer is NIL and the offset and limit are not equal, then **Error[invalidParameter]** is raised.

67.2.16 Appending to Writers

The operations in this section append to writers. When there is insufficient space in the writer to hold the bytes to be appended, the operations attempt to allocate from the writer's zone a new byte sequence of sufficient size. If there is insufficient space and the writer's zone is NIL, the signal **InsufficientRoom** is raised. If it is resumed, the operations will append as many characters as will fit. Errors resulting from allocating from the zone are allowed to propagate. An expanded set of appending operations are available using the **XFormat** interface.

```
AppendReader: PROCEDURE [
   to: Writer, from: Reader, fromEndContext: Context ← unknownContext,
   extra: CARDINAL ← 0];
```

AppendReader appends the reader to the writer. If either the reader or writer is NIL, this operation simply returns. The end context of the writer is updated to **fromEndContext** if it is not **unknownContext** and **ComputeEndContext[r]** otherwise. The signal **InsufficientRoom** may be raised as described above.

```
AppendChar: PROCEDURE [to: Writer, c: Character, extra: CARDINAL ← 0];
```

AppendChar appends the character to the writer. If the writer is NIL, this operation simply returns. The signal **InsufficientRoom** may be raised as described above. If it is resumed, nothing is appended.

```
AppendStream: PROCEDURE [
   to: Writer, from: Stream.Handle, nBytes: CARDINAL,
   fromContext: Context ← unknownContext, extra: CARDINAL ← 0]
   RETURNS [bytesTransferred: CARDINAL];
```

AppendStream appends **nBytes** from the stream **from** to the writer. If either the stream or writer is NIL, this operation simply returns. The end context of the writer is updated to **fromEndContext** if it is not **unknownContext** and **ComputeEndContext[r]** otherwise. The signal **InsufficientRoom** may be raised as described above. If it is resumed, as much of the reader as will fit in the space available is appended. **AppendStream** returns the actual number of bytes transferred.

Note: There is currently a bug in the interface such that the **fromContext** parameter is defaulted to **vanillaContext** instead of **unknownContext**.

```
AppendSTRING: PROCEDURE [
   to: Writer, from: LONG STRING, homogeneous: BOOLEAN ← FALSE, extra: CARDINAL ← 0];
```

AppendSTRING appends the string to the writer. If either the string or writer is NIL, this operation simply returns. The end context of the writer is updated to **vanillaContext** if **homogeneous** is TRUE, otherwise its value is computed from the parameter **from**. The signal **InsufficientRoom** may be raised as described above.

```
InsufficientRoom: SIGNAL [needsMoreRoom: Writer, amountNeeded: CARDINAL];
```

The signal **InsufficientRoom** is raised by the append operations when the writer does not have enough room to contain the appendee and the writer's zone is NIL. Resuming this signal will result in as much as possible being appended.

```
AppendExtensionIfNeeded: PROCEDURE [to: Writer, extension: Reader]
   RETURNS [didAppend: BOOLEAN];
```

AppendExtensionIfNeeded checks to see if there is a period somewhere in the writer other than the last character. If there is, FALSE is returned. If not, it appends a period if the writer does not already end in one, then appends **extension** and returns TRUE. **AppendChar** and **AppendReader** are used to append and they may raise **InsufficientRoom**.

67.2.17 Editing Writers

```
Piece: PROCEDURE [r: Reader, firstChar, nChars: CARDINAL]
  RETURNS [piece: ReaderBody, endContext: Context];
```

Piece returns a reader body that describes the **firstChar** through **firstChar + nChars** logical characters of **r**. **piece** will describe as many characters of **r** that are in that range, possibly none if **CharacterLength[r]** is less than or equal to **firstChar**. The context of the last character of **piece** is also returned. It may raise the **InvalidEncoding** error.

```
ReplacePiece: PROCEDURE [
  w: Writer, firstChar, nChars: CARDINAL, r: Reader,
  endContext: Context ← unknownContext];
```

ReplacePiece is an editing operation for writers. It replaces **nChars** of **w** starting at **firstChar** with the characters of **r**. **nChars** may be zero and **r** may be empty. If the reader is not empty, **endContext** is needed to update the end context of the writer if the piece replacement is at the end, or to determine if there needs to be a character set shift between the bytes of **r** and the (**firstChar + nChar**)th character of **w**. If **endContext** is **unknownContext** it will be computed from **r**. The signal **InsufficientRoom** may be raised as described above if the operation resulted in a net addition of bytes. **ReplacePiece** may raise the **InvalidEncoding** error.

67.2.18 Conversion from Readers

```
Block: PROCEDURE [r: Reader] RETURNS [block: Environment.Block, context: Context] ;
```

Block returns both a block that describes the bytes in **r** as well as the context of **r**. This operation may be used by clients that need to examine the bytes directly. **Note:** The bytes of the block should not be written.

```
NSStringFromReader: PROCEDURE [r: Reader, z: UNCOUNTED ZONE]
  RETURNS [ns: NSString.String] ;
```

NSStringFromReader creates an **NSString.String** from a reader. It always copies the bytes of the reader into a new allocation unit allocated from the zone. The resulting string should be deallocated using operations from the **NSString** interface. Errors from allocating from the zone are allowed to propagate.

67.2.19 Reverse Character Operations

```
ReverseMap: PROCEDURE [r: Reader, proc: MapCharProc] RETURNS [c: Character];
```

ReverseMap is similar to **Map**, except it enumerates the characters in reverse order. It is less efficient than **Map** because encoding characters makes backward scanning difficult. It may raise the **InvalidEncoding** error.

```
ReverseLop: PROCEDURE [
  r: Reader, endContext: LONG POINTER TO Context,
   backScan: BackScanClosure ← [NIL, NIL]]
   RETURNS [c: Character];

BackScanClosure: TYPE = RECORD [proc: BackScanProc, env: LONG POINTER];

BackScanProc: TYPE = PROCEDURE [beforePos: CARDINAL, env: LONG POINTER]
   RETURNS [pos: CARDINAL, context: Context];
```

ReverseLop is similar to Lop, except it takes characters off the end of the reader. It is less efficient than Lop because encoding characters makes backward scanning difficult. If the reader is empty, it returns XChar.not. If endContext↑ is not unknownContext, then it must be correct. It may be changed by a call to ReverseLop. If ReverseLop backs up over a character set shift, it will set endContext↑ to unknownContext. It may raise the InvalidEncoding error.

The BackScanClosure and BackScanProc provide a way for the client to inform ReverseLop of the context in effect before a character set shift. If endContext↑ is unknownContext, then backScan.proc is called with a byte offset before which it desires a context and backScan.env. It should return a context for some position before the passed one, as well as the actual position corresponding to that context. Simple clients of ReverseLop need not provide a BackScanClosure. It is provided for clients that have information about location of character set changes within the reader.

67.3 Usage/Examples

67.3.1 Designing Interfaces with Readers

Designing interfaces to use readers is more complicated than LONG STRINGs. The biggest complication is the two-level allocation scheme involving readers → reader bodies → bytes. In most cases, the bytes are relatively static and are relatively easy to deal with. The main problem is determining whether to use a reader or a reader body. It helps to keep in mind the following guideline: *keep reader bodies to describe the bytes*. Save a pointer to a string by putting the reader body, not the reader, in the data structure. This way, one doesn't have to worry about who owns the storage for the reader body. Consider the following interface fragment:

```
Handle: TYPE = LONG POINTER TO Object;

Object: TYPE = RECORD [
   next: Handle,
   count: CARDINAL,
   name: XString.ReaderBody];

AddAnother: PROCEDURE [name: XString.Reader];
```

Instead of storing the name in the object as a reader, it is stored as a reader body. A reader is quickly generated when needed by the expression @h.name.

Another guideline is for a procedure to *take a reader and return a reader body*. The idea is that passing readers as parameters reduces the number of words of parameters. Returning reader bodies allows the client to manage the storage for the reader body.

A third guideline is that *clients should be able to pass pointers to local reader bodies*. If an implementation kept strings passed to it by saving readers, clients would have to allocate the reader body from permanent storage, not from the local frame. If implementations keep reader bodies instead of readers, passing **@localReaderBody** will not result in dangling pointers. For example, consider the following fictional procedure that renames a file:

```
RenameFile: PROCEDURE [oldName:XString.Reader] = {
   rb: XString.ReaderBody;
   rb ← SomeInterface.GetNewName[];
   file ← SomeInterface.LookupByName[oldName];
   SomeInterface.Rename[file: file, newName: @rb]};
```

The procedure **RenameFile** takes a reader that it simply passes to another operation. While taking a reader body is equilvalent, it is more efficient to take a reader, particularily when strings are just being passed around. The operation **GetNewName** returns a reader body. If it returned a reader, it would have to define where the storage for the reader body was kept. Either it would have to be global, or it would have to be deallocated from a known place after **RenameFile** was done with it. It is just simplier to return reader bodies than to deal with the allocation problems of reader bodies. It is hard enough to make sure ownership of the bytes is handled correctly. The new name to the **Rename** operation is a pointer to a local reader body. **Rename** should copy the reader body (and the bytes) if it intends to save the characters.

Warning: Designing interfaces that do not allow passing pointers to local reader bodies should be avoided.

67.3.2 Using Readers

One of the simple things to do with strings is to pass string literals. Because there is no compiler support for **XString**, it is harder to do. The code fragment below gives an example of how to pass string literals:

```
GetUserCmFile: PROCEDURE RETURNS [file: SomeInterface.FileHandle] = {
   rb: XString.ReaderBody ← XString.FromSTRING["User.cm"L];
   file ← SomeInterface.LookupByName[name: @rb]};
```

Looking at all the characters in a string for something like switch processing is another common operation:

```
Options: TYPE = RECORD [debug, verify, start: BOOLEAN ← FALSE];

ParseSwitches1: PROCEDURE [r: XString.Reader] RETURNS [options: Options] = {
   rb: XString.ReaderBody ← XString.Dereference[r];
   c: XString.Character;
   sense: BOOLEAN ← TRUE;
   WHILE (c ← XString.Lop[@rb]) # XChar.not DO
      SELECT c FROM
         'd.ORD = > {options.debug ← sense; sense ← TRUE};
         'v.ORD = > {options.verify ← sense; sense ← TRUE};
         's.ORD = > {options.start ← sense; sense ← TRUE};
         '-.ORD = > sense ← FALSE;
         ENDCASE = > sense ← TRUE;
      ENDLOOP;
   RETURN};
```

ParseSwitches1 uses **Lop** to look at each character of **r** and set the appropriate option. Because **Lop** changes the reader body to remove the first character, **ParseSwitches1** uses **Dereference** to copy the reader body to a local variable.

The operations **Lop**, **Run**, **ScanForCharacter**, and **Scan** all update the reader body of their reader parameter. If the reader body must not be altered, it should be copied as in the above example.

```
ParseSwitches2: PROCEDURE [r: XString.Reader] RETURNS [options: Options] = {
   sense: BOOLEAN ← TRUE;
   proc: XString.MapCharProc = {
      SELECT c FROM
         'd.ORD = > {options.debug ← sense; sense ← TRUE};
         'v.ORD = > {options.verify ← sense; sense ← TRUE};
         's.ORD = > {options.start ← sense; sense ← TRUE};
         '-.ORD = > sense ← FALSE;
         ENDCASE = > sense ← TRUE;
      RETURN[stop: FALSE]};
   [] ←XString.Map[r: r, proc: proc];
   RETURN};
```

ParseSwitches2 uses **Map** to look at each character of **r** and set the appropriate option.

67.3.3 Simple Parser Example

Below is a simple program that accepts a sequence of characters from a procedure and parses them into tokens. It collects characters one a a time and appends them to the writer buffer. If the string of characters is empty, it returns a keyword token of *invalid*. If the first character is a digit, it returns a number token, converting the string into the number. Otherwise it compares the string with the four keywords. If it is not a keyword, it copies the string from the buffer to a new reader and returns the id token.

```
-- Example.mesa

DIRECTORY
   XString USING [
      AppendChar, Character, ClearWriter, Empty, Equal, First, InvalidNumber,
      NewWriterBody, OverFlow, Reader, ReaderBody, ReaderFromWriter,
      ReaderToNumber, WriterBody];

Example: PROGRAM IMPORTS XString = {
   OPEN XString;

TokenClass: TYPE = {keyword, id, number};
Keyword: TYPE = {begin, end, do, endloop, eof, invalid};
Token: TYPE = RECORD [
   SELECT class: TokenClass FROM
      keyword = > [keyword: Keyword],
      id = > [id: ReaderBody],
        number => [number: LONG INTEGER],
        ENDCASE];

Input: TYPE = PROCEDURE RETURNS [Character];

eof, space: Character = ... ;
keywords: LONG DESCRIPTOR FOR ARRAY Keyword[begin..endloop] OF ReaderBody = ... ;
zone: UNCOUNTED ZONE ←... ;
buffer: WriterBody ← NewWriterBody[maxLength: 40, z: zone] ;

Parse: PROCEDURE [input: Input] RETURNS [token: Token] = {
    r: Reader ← ReaderFromWriter[@buffer];
    c: Character;
    ClearWriter[buffer];
    DO
        SELECT (c ← input[]) FROM
            space => EXIT;
            eof => RETURN[[keyword[IF Empty[r] THEN invalid ELSE eof]]];
            ENDCASE => AppendChar[@buffer, c];
        ENDLOOP;
    IF Empty[r] THEN RETURN[[keyword[invalid]];
    IF First[r] IN ['0.ORD..'9.ORD] THEN {
        token ← [number[ReaderToNumber[r, 10 !
            InvalidNumber, Overflow => {token ← [keyword[invalid]]; CONTINUE}]]];
        RETURN};
    SELECT TRUE FROM
        Equal[r, @keywords[begin]] => token ← [keyword[begin]];
        Equal[r, @keywords[end]] => token ← [keyword[end]];
        Equal[r, @keywords[do]] => token ← [keyword[do]];
        Equal[r, @keywords[endloop]] => token ← [keyword[endloop]];
        ENDCASE =>
            token ← [id[CopyToNewReaderBody[r: r, z: zone]]];
        RETURN};

}...
```

67.4 Index of Interface Items

Item	Page
AppendChar: PROCEDURE	13
AppendExtensionIfNeeded: PROCEDURE	13
AppendReader: PROCEDURE	13
AppendStream: PROCEDURE	13
AppendSTRING: PROCEDURE	13
BackScanClosure: TYPE	15
BackScanProc: TYPE	15
Block: PROCEDURE	14
BreakCharOption: TYPE	11
BreakTable: TYPE	11
BreakTableObject: TYPE	11
Byte: TYPE	4
ByteLength: PROCEDURE	5
Bytes: TYPE	4
ByteSequence: TYPE	3
Character: TYPE	5
CharacterLength: PROCEDURE	5
ClearWriter: PROCEDURE	8
Compare: PROCEDURE	10
CompareStringsAndStems: PROCEDURE	10
ComputeEndContext: PROCEDURE	12
Context: TYPE	2
CopyReader: PROCEDURE	7
CopyToNewReaderBody: PROCEDURE	7
CopyToNewWriterBody: PROCEDURE	9
Dereference: PROCEDURE	5
DescribeReader: Courier.Description	12
DescribeReaderBody: Courier.Description	12
Empty: PROCEDURE	5
emptyContext: Context	3
Equal: PROCEDURE	9
Equivalent: PROCEDURE	9
Error: ERROR	6
ErrorCode: TYPE	6
ExpandWriter: PROCEDURE	9
First: PROCEDURE	5
FreeReaderBytes: PROCEDURE	7
FreeWriterBytes: PROCEDURE	9
FromBlock: PROCEDURE	7
FromChar: PROCEDURE	7
FromNSString: PROCEDURE	7
FromSTRING: PROCEDURE	7
InitBreakTable: PROCEDURE	12
InsufficientRoom: SIGNAL	13
InvalidEncoding: ERROR	6
InvalidNumber: SIGNAL	10
Lop: PROCEDURE	5
Map: PROCEDURE	6
MapCharProc: TYPE	6
NewWriterBody: PROCEDURE	8
NSStringFromReader: PROCEDURE	14
NthCharacter: PROCEDURE	5
nullReaderBody: ReaderBody	4
nullWriterBody: WriterBody	4
Overflow: SIGNAL	11
Piece: PROCEDURE	14
Reader: TYPE	3
ReaderBody: TYPE	3
ReaderFromWriter: PROCEDURE	6
ReaderInfo: PROCEDURE	5
ReaderToNumber: PROCEDURE	10
ReaderToNumberX: PROCEDURE	10
ReadOnlyBytes: TYPE	3
Relation: TYPE	10
ReplacePiece: PROCEDURE	14
ReverseLop: PROCEDURE	15
ReverseMap: PROCEDURE	14
Run: PROCEDURE	12
Scan: PROCEDURE	11
ScanForCharacter: PROCEDURE	11
SortOrder: TYPE	9
StopOrNot: TYPE	11
unknownContext: Context	3
ValidateReader: PROCEDURE	12
vanillaContext: Context	3
Writer: TYPE	4
WriterBody: TYPE	4
WriterBodyFromBlock: PROCEDURE	8
WriterBodyFromNSString: PROCEDURE	8
WriterBodyFromSTRING: PROCEDURE	8
WriterInfo: PROCEDURE	8

68

XTime

68.1 Overview

The **XTime** interface provides functions for acquiring and editing times into strings or strings into times. It provides the same function as the XDE **Time** interface but deals with **XString.Readers** instead of **LONG STRINGs**.

68.2 Interface Items

68.2.1 Acquiring Time

```
Current: PROCEDURE RETURNS [time: System.GreenwichMeanTime];
```

Current returns the current time.

```
ParseReader: PROCEDURE [
  r: XString.Reader, treatNumbersAs: TreatNumbersAs ← dayMonthYear]
  RETURNS [time:System.GreenwichMeanTime, notes: Notes, length: CARDINAL];

ParseWithTemplate: PROCEDURE [r, template: XString.Reader]
  RETURNS [time: System.GreenwichMeanTime, notes: Notes, length: CARDINAL];

TreatNumbersAs: TYPE = {dayMonthYear, monthDayYear, yearMonthDay,
  yearDayMonth, dayYearMonth, monthYearDay};
```

The **ParseReader** procedure parses the reader **r** and returns a GMT **time** according to the Pilot standard. **treatNumbersAs** indicates how to interpret **r**. **ParseWithTemplate** parses **r** according to **template**. **template** serves as an interpreter for deriving time fields from **r** (see § 68.3). The date syntax is a somewhat less restrictive version of RFC733; full RFC733 is recognized, plus forms like "month day, year", "mm/dd/yy", and variations with Roman numerals used for the month. The form "year month day" is also accepted if the year is a full four-digit quantity. Forms with "-" instead of significant space are also acceptable, as well as forms in which a delimiter (space or "-") can be elided without confusion. The time is generally assumed to be in RFC733 format, optionally including a time zone specification. In addition, am or pm may appear following the time (but preceding the time zone, if any). **notes** is interpreted as described below. **length** indicates the number of characters consumed by the parser (that is, it is the index of the first character of the argument that was not examined by the parser). This procedure can raise the error **Unintelligible**.

Notes: TYPE = {normal, noZone, zoneGuessed, noTime, timeAndZoneGuessed};

Notes is used as one of the return values from the call on **ParseReader**. **normal** means the value returned is unambiguous; **noZone** means that a time-of-day without a time zone indication was present. (The local time zone as provided by **System.LocalTimeParameters** is assumed.) **zoneGuessed** is returned instead of **noZone** if local time parameters are not available; the time zone is assumed to be Pacific Time (standard or daylight time is determined by the date). **noTime** and **timeAndZoneGuessed** are equivalent to **noZone** and **zoneGuessed** respectively, where the time is assumed to be 00:00:00 (local midnight).

Unintelligible: ERROR [vicinity: CARDINAL];

If **ParseReader** cannot reasonably interpret its input as a date, **Unintelligible** is raised; **vicinity** gives the approximate index in the input string where the parser gave up.

68.2.2 Editing Time

```
Append: PROCEDURE [
   w: XString.Writer, time: System.GreenwichMeanTime ← defaultTime,
   template: XString.Reader ← dateAndTime, ltp: LTP ← useSystem];
```

Append appends the **time** in human-readable form to **w**. **template** determines which fields are appended. **ltp** provides the local time parameters (discussed below).

```
Format: PROCEDURE [
   xfh: XFormat.Handle ← NIL, time: System.GreenwichMeanTime ← defaultTime,
   template: XString.Reader ← dateAndTime, ltp: LTP ← useSystem];
```

Format converts **time** to a string by calling **XTime.Append** using **template** to specify which template to use. **xfh.proc** is then called. If **time** is defaulted, the current time is used.

```
Pack: PROCEDURE [unpacked: Unpacked, useSystemLTP: BOOLEAN ← TRUE]
   RETURNS [time: System.GreenwichMeanTime];
```

Pack converts **unpacked** into the Pilot-standard **System.GreenwichMeanTime**. **useSystemLTP** indicates that **Pack** should use the system's parameters. If the local time parameters are not available to Pilot, **System.LocalTimeParametersUnknown** is raised. If **unpacked** is invalid, **Invalid** is raised.

Packed: TYPE = System.GreenwichMeanTime;

Packed is retained for compatability.

```
Unpack: PROCEDURE [
   time: System.GreenwichMeanTime ← defaultTime, ltp: LTP ← useSystem]
   RETURNS [unpacked: Unpacked];
```

Unpack converts **time** into its unpacked representation. If **time** is defaulted, the current time is used. **ltp** provides local time parameters. If the local time parameters are not available to Pilot, System.LocalTimeParametersUnknown is raised. If time is invalid, Invalid is raised.

```
Unpacked: TYPE = RECORD[
  year: [0..2104], month: [0..12), day: [0..31],
  hour: [0..24), minute: [0..60), second: [0..60),
  weekday: [0..6], dst: BOOLEAN, zone: System.LocalTimeParameters];
```

Unpacked values record dates by their pieces. The fields are filled by Unpack, described above, which operates on the time and date as kept internally by Pilot. year = 0 corresponds to 1968. For month, January is numbered 0, 1,and so forth. Days of the month have their natural assignments. For weekday, Monday is numbered 0. dst indicates Daylight Standard Time. zone indicates time zones.

```
LTP: TYPE = RECORD[
  r: SELECT t: * FROM
    useSystem = > [],
    useThese = > [lpt: System.LocalTimeParameters]]
  ENDCASE];
```

LTP passes local time parameters to several procedures. Usually they are defaulted to the system's parameters.

```
Invalid: ERROR;
```

68.2.3 Useful Constants and Variables

```
dateAndTime: XString.Reader;
dateOnly: XString.Reader;
timeOnly: XString.Reader;
```

These variables are templates that are supplied by XComSoftMessage for use in the Append operation.

```
defaultTime: System.GreenwichMeanTime = System.gmtEpoch;
```

defaultTime always means the current time.

```
useSystem: useSystem LTP = [useSystem[]];
useGMT: useThese LTP = [useThese[[west, 0, 0, 0, 0]]];
```

These local time parameters are exported for client convenience.

68.3 Usage/Examples

68.3.1 ParseReader Template Definitions

The template for times is a reader with fields, using the standard definition of naming fields (that is, a number enclosed by angle brackets). The definition of the fields for times are:

<1>	Month as a number*
<2>	Day as a number*
<3>	Year as a four-digit number
<4>	Year as a two-digit number
<5>	Month name
<6>	Month name with a maximum of three characters
<7>	Hour as digits in range [0..12)*
<8>	Hour as digits in range [0..24)*
<9>	Minutes, always two digits, zero-filled (e.g., 23, 04)
<10>	Seconds, always two digits, zero-filled (e.g., 23, 04)
<11>	Day of week
<12>	Time zone
<13>	AM or PM

* If the number begins with a 0, the number is zero-filled to two digits.

Examples

<2>-<6>-<4> <7>:<9>	18-Nov-83 8:36
<2> <5> <3>	18 November 1983
<2>-<6>-<4> <7>:<9>:<10> <12> (<11>)	18-Nov-83 08:36:42 PST (Friday)
<7>:<9> <13>	8:36 AM
<08><9> hrs	0836 hrs

68.3.2 Example

```
-- Data structure to record time in both Packed and Unpacked form.
Data: TYPE = RECORD [
   startTime: XString.ReaderBody ← XString.nullReaderBody,
   endTime: XString.ReaderBody ← XString.nullReaderBody,
   pStartTime: XTime.Packed ← System.gmtEpoch,
   pEndTime: XTime.Packed ← System.gmtEpoch];

-- Retrieves, unpacks, stores, and displays the time.
GetAndDisplayTime: PROC[packTime: XTime.Packed] = {
   time: XTime.Unpacked ← XTime.Unpack[packTime];
   TimeDisplay [time.year, time.month, time.day];
   };

ParseTimes: PROC[data: Data] = {
   data.pStartTime ← XTime.ParseReader[@data.startTime !
      XTime.Unintelligible => Error[BadStartTime, vicinity]].time;
   data.pEndTime ← XTime.ParseReader[@data.endTime !
      XTime.Unintelligible => Error[BadEndTime, vicinity]].time;
   };

-- Parses time into an XString.ReaderBody
PackedToString: PROC[time: System.GreenwichMeanTime]
```

```
RETURNS [rb:XString.ReaderBody ← XString.nullReaderBody] = {
template: XString.ReaderBody ← XString.FromSTRING[
   "<2>-<6>-<4> <8>:<9>"L];
wb: XString.WriterBody ← XString.NewWriterBody[24, zone];
XTime.Append[ @wb, time, @template ];
rb ← XString.CopyToNewReaderBody[XString.ReaderFromWriter[@wb], zone];
XString.FreeWriterBytes[@wb];
};
```

68.4 Index of Interface Items

69

XToken

69.1 Overview

The **XToken** interface provides general scanning and simple parsing facilities for collecting tokens from a character input stream.

The basic data structure is the **Object**, which encapsulates the source of characters to be parsed. It contains a procedure that returns the next character in the input stream and the final character that is read from the input stream.

The basic operations collect characters from the input stream into tokens. Clients can define arbitrary token classes by using filters. Clients can define their own filters or use one of the standard filters provided by **XToken**. Frequently, some portion of the input stream, such as blanks, are only delimiters and are usually skipped when collecting a token. The type **SkipMode** defines the options for skipping characters. Quoted tokens are a feature provided by **XToken**. By using procedures to define opening- and closing-quote characters, **XToken** allows the client to define a large number of quoting schemes. Several common quote procedures are supplied.

XToken provides operations that collect standard tokens such as boolean and numeric values. It also provides built-in handles that understand **XString.Readers** and **Stream.Handles** as sources of characters.

Operations that return a reader body allocate their storage from the implementation's own heap. Clients should call **FreeTokenString** to release this storage.

69.2 Interface Items

69.2.1 Character Source Definitions

```
Handle: TYPE = LONG POINTER TO Object;

Object: TYPE = MACHINE DEPENDENT RECORD [
   getChar(0):GetCharProcType, break(1): XChar.Character ← 0];

GetCharProcType: TYPE = PROCEDURE [h: Handle] RETURNS [c: XChar.Character];
```

The **Object** encapsulates the source of characters to be parsed. The **XToken** package uses the **getChar** field to obtain the stream of characters. It assumes that the source has been exhausted when **getChar** returns **XChar.null** or **XChar.not**. **XToken** uses the **break** field to record the final character that it reads. It records the final character because there is no way to put back a character into the character source. It must read one character beyond the token it is parsing to ensure that it has reached the end of the input. If it simply returned the token, this character would be lost. Since the **XToken** package stores the last character in the **Object**, that character is available to the client. The client can ignore it, inspect it to decide what to parse next, or put it back into the character source.

A **GetCharProcType** provides a stream of characters to be parsed. It should return either **XChar.null** or **XChar.not** when the stream of characters has been exhausted. The **Handle** is passed into the **GetCharProcType** so that a client can hide instance data in his object. Although there is not an instance data field in **Object**, the client could LOOPHOLE a pointer to a larger record that contained its data.

69.2.2 Filter Definitions

```
FilterProcType: TYPE = PROCEDURE [
  c: XChar.Character, data: FilterState] RETURNS [inClass: BOOLEAN];
```

```
FilterState: TYPE = LONG POINTER TO StandardFilterState;
```

```
StandardFilterState: TYPE = ARRAY [0..2) OF UNSPECIFIED;
```

A **FilterProcType** is the mechanism by which a client defines classes of characters. Procedures that use filters will call them once for each candidate character. The filter should return TRUE if the character is in the class and FALSE otherwise. The **FilterState** permits the filter to maintain the state of the parse. Operations that require a **FilterProcType** and **FilterState**, will initialize the **StandardFilterState** to ALL[0]. If the filter requires filter state but **data** is NIL, the signal **NilData** should be raised.

Some clients' filters may need more than two words of state for their filter. In that case they should define a record that first contains a **StandardFilterState**, then define the additional space they need, and then loophole the filter state to a pointer to the record they defined.

69.2.3 Skip Mode Definitions

```
SkipMode: TYPE = {none, whiteSpace, nonToken};
```

SkipMode controls what characters an operation will skip before collecting a token.

none	The term **none** means no characters should be skipped, and the token should start with the next character.
whiteSpace	The term **whiteSpace** means characters (space, carriage return, and tab) should be skipped before collecting the token.
nonToken	The term **non Token** means any characters that are not legal token characters should be skipped before collecting the token.

69.2.4 Quoted Token Definitions

QuoteProcType: TYPE = PROCEDURE [
c: XChar.Character] RETURNS [closing: XChar.Character];

nonQuote: XChar.Character = . . . ;

QuoteProcType defines the procedure used by **MaybeQuoted** to recognize quoted tokens. If **c** is a quote character, it should return the corresponding closing-quote character. If **c** is not a quote character, it should return **nonQuote**.

69.2.5 Built-in Handles

ReaderToHandle: PROCEDURE [r: XString.Reader]
RETURNS [h: Handle];

ReaderToHandle creates a **Handle** whose source of characters are the characters in **r**. The bytes of **r** are not copied, so clients are responsible for synchronizing access to the reader with the **XToken** package.

FreeReaderHandle: PROCEDURE [h: Handle] RETURNS [nil: Handle];

FreeReaderHandle destroys a **Handle** created by **ReaderToHandle**. It does not destroy the underlying reader. It returns **NIL**.

StreamToHandle: PROCEDURE [s: Stream.Handle] RETURNS [h: Handle];

StreamToHandle creates a **Handle** whose source of characters is a stream. The stream should signify the end of characters by raising the signal **Stream.EndOfStream**.

FreeStreamHandle: PROCEDURE [h: Handle] RETURNS [s: Stream.Handle];

FreeStreamHandle destroys a **Handle** created by **StreamToHandle**. It returns the underlying stream.

69.2.6 Boolean and Numeric Tokens

Boolean: PROCEDURE [h: Handle, signalOnError: BOOLEAN ← TRUE] RETURNS [true: BOOLEAN];

Boolean parses the next characters of the source as a boolean constant. Valid Boolean values are "TRUE" or "FALSE," but unlike the Mesa language, case does not matter ("true" and "false" are also acceptable). In case of a syntax error, the signal **SyntaxError** is optionally raised. If **signalOnError** is **FALSE**, or **SyntaxError** is resumed, then **FALSE** is returned for a syntax error. This procedure skips leading white space.

Number: PROCEDURE [h: Handle, radix: CARDINAL, signalOnError: BOOLEAN ← TRUE]
RETURNS [u: LONG UNSPECIFIED];

Number parses the next characters of the source as a number in radix **radix**. Numbers have the format specified in **XString.ReaderToNumber**. In case of a syntax error, the signal

SyntaxError is optionally raised. If **signalOnError** is FALSE, or **SyntaxError** is resumed, zero is returned for a syntax error. This procedure skips leading white space.

```
NumberX: PROCEDURE [h: Handle, radix: CARDINAL, signalOnError: BOOLEAN ← TRUE,
   numberParms: XDigits.NumberParms ← XDigits.numberParms]
   RETURNS [u: LONG UNSPECIFIED];
```

NumberX is the same as **Number** except that **numberParms** can be specified. See the XDigits chapter for more details. If **radix** is other than 10, **numberParms.representation** and **numberParms.thousandsSep** will be ignored. Fine Point: This procedure is currently exported through **XTokenX**.

```
Decimal: PROCEDURE [
   h: Handle, signalOnError: BOOLEAN ← TRUE] RETURNS [i: LONG INTEGER];
```

Decimal is just like **Number**, but with a **radix** of 10.

```
DecimalX: PROCEDURE [
   h: Handle, signalOnError: BOOLEAN ← TRUE,
   numberParms: XDigits.NumberParms ← XDigits.numberParms]
   RETURNS [i: LONG INTEGER];
```

DecimalX is just like **NumberX**, but with a **radix** of 10. See the XDigits chapter for more details. Fine Point: This procedure is currently exported through **XTokenX**.

```
Octal: PROCEDURE [
   h: Handle, signalOnError: BOOLEAN ← TRUE] RETURNS [c: LONG CARDINAL];
```

Octal is just like **Number**, but with a **radix** of 8.

69.2.7 Basic Token Routines

```
Filtered: PROCEDURE [
   h: Handle, data: FilterState, filter: FilterProcType, skip: SkipMode ← whiteSpace,
   temporary: BOOLEAN ← TRUE]
   RETURNS [value: XString.ReaderBody];
```

Filtered collects the token string defined by the client's filter. If the client-instance data parameter **data** is not NIL, the first two words of **data** are set to zero before any calls are made to **filter**. **filter** is called with **data** once on each character until it returns FALSE. The string returned, which may be **XString.nullReaderBody**, must be freed by calling **FreeTokenString**. Leading characters are skipped according to the value of **skip**. If **temporary** is TRUE, it is assumed that the string will be freed shortly, and no effort is made to use the minimum storage for it. If **temporary** is FALSE, the minimum amount of storage is used. **filter** may raise **NilData**.

```
FreeTokenString: PROCEDURE [s: XString.Reader] RETURNS [nil: XString.Reader ← NIL];
```

FreeTokenString frees bytes of the reader. It is used to free the strings allocated by **Filtered**, **Item**, and **MaybeQuoted**. It returns NIL.

```
Item: PROCEDURE [
   h: Handle, temporary: BOOLEAN ← TRUE] RETURNS [value: XString.ReaderBody];
```

Item returns the next token delimited by white space. Leading white space is skipped and the characters are collected until another white-space character is encountered. The string returned must be freed by calling **FreeTokenString**. If **temporary** is TRUE, it is assumed that the string will be freed shortly, and no effort is made to use the minimum storage for it. If **temporary** is FALSE, only as much storage is used for the string as is needed.

```
MaybeQuoted: PROCEDURE [
   h: Handle, data: FilterState, filter: FilterProcType ← NonWhiteSpace,
   isQuote: QuoteprocType ← Quote, skip: SkipMode ← whiteSpace,
   temporary: BOOLEAN ← TRUE]
   RETURNS [value: XString.ReaderBody];
```

MaybeQuoted returns the next quoted token. The first candidate character is passed to **isQuote**, which either returns **nonQuote** or the closing-quote character. If a closing-quote character other than **nonQuote** is returned, characters are collected in the token until the closing quote is encountered. If the input is exhausted before the closing quote is encountered, the signal **UnterminatedQuote** will be raised. If it is resumed, **MaybeQuoted** returns the token collected up until that point. The closing-quote character may be included in the token by including two instances of the character in the input; that is, if **MaybeQuoted** encounters two closing-quote characters in a row, it will insert one closing-quote character in the token rather than terminating the token on the first closing quote. The outer quote characters are not part of the token and are discarded. If **nonQuote** is returned from the **isQuote** procedure, the filter is used to collect characters the same way it is used in **Filtered**: **filter** is called with client-instance data parameter **data** once on each character until it returns FALSE. In either case (quoted or filtered), the break character returned in the **Handle** is the character following the token.

Leading characters are skipped according to the value of **skip**.

If **temporary** is TRUE, it is assumed that the string will be freed shortly and no effort is made to use the minimum storage for it. If **temporary** is FALSE, only as much storage is used for the string as is needed. The string returned must be freed by calling **FreeTokenString**.

```
Skip: PROCEDURE [
   h: Handle, data: FilterState, filter: FilterProcType, skipInClass: BOOLEAN ← TRUE];
```

Skip is used to skip over characters. A filter is provided to define the class of characters, and the boolean **skipInClass** indicates whether the characters to be skipped are those accepted by the filter or those rejected by it. If the client-instance data parameter **data** is not NIL, the first two words of **data** are set to zero before any calls are made to **filter**. If **data** is NIL and **filter** references data, the signal **NilData** should be raised.

69.2.8 Signals and Errors

```
SyntaxError: SIGNAL [r: XString.Reader];
```

The resumable SIGNAL **SyntaxError** can be raised if incorrect syntax is encountered by **Boolean**, **Decimal**, **Number**, or **Octal**. In each case, resuming the signal causes the procedure to return a default value (described in the discussion of the various procedures). The reader parameter is the token collected that has the wrong syntax.

NilData: SIGNAL;

Procedures that take a **FilterProcType** argument also take an argument that is a pointer to client instance data. If the client has no need for instance data, it can pass a **NIL** as the instance data pointer. If a **FilterProcType** attempts to access the client-instance data, but the client passed in **NIL** instead of a pointer to instance data, the signal **NilData** should be raised. Implementors of **FilterProcTypes** are strongly encouraged to check for **NIL** and raise this condition if they use client-instance data.

UnterminatedQuote:SIGNAL;

The resumable **SIGNAL UnterminatedQuote** is raised from **MaybeQuoted** if the **getChar** procedure of the **Handle** returns **XChar.not** or **XChar.null** before the terminating quote character has been read. If the signal is resumed, **MaybeQuoted** will return as if it had read a closing-quote character.

69.2.9 Built-in Filters

Alphabetic: FilterProcType;

Alphabetic defines the class of alphabetic characters; that is, the characters **'a** through **'z** and **'A** through **'Z**. This procedure requires no filter state.

AlphaNumeric: FilterProcType;

AlphaNumeric defines the class of alphanumeric characters, that is, the characters **'a** through **'z**, **'A** through **'Z**, and **'0** through **'9**. This procedure requires no filter state.

Delimited: FilterProcType;

When **Delimited** is passed to a procedure such as **Filtered**, the value of **skip** passed along with it must be **nonToken**. It will skip leading white space, then define the first character of the token to be both the opening-quote character and the closing-quote character, returning all characters occurring between the first and second appearance of that character.

Line: FilterProcType;

Line defines a class containing all characters except the carriage return. It can be used to collect a line of information. This procedure requires no filter state.

NonWhiteSpace: FilterProcType;

NonWhiteSpace FilterProc defines all characters that are not white space; that is, **WhiteSpace[char] = ~NonWhiteSpace[char]**. This procedure requires no client data (**data** may be **NIL**.)

Numeric: FilterProcType;

Numeric defines the class of numeric characters (the characters '0 through '9) This procedure requires no filter state.

Switches: FilterProcType;

Switches can be used to collect switch characters. It accepts the characters '~, '-, and alphanumeric characters. This procedure requires no filter state.

WhiteSpace: FilterProcType;

The **WhiteSpace FilterProcType** defines the white space characters. This filter is used by **Token** for skipping white space. This procedure requires no filter state.

69.2.10 Built-in Quote Procedures

Brackets: QuoteProcType;

Brackets recognizes the following sets of matching open/close quote pairs: ( ), [ ], { }, and < >.

Quote: QuoteProcType;

Quote recognizes single quote and double quote.

69.3 Usage/Examples

69.3.1 Collecting Tokens

The following example collects name and number pairs from a stream. It uses the built-in stream handle provided by **XToken** for the source of characters. It uses the **Item** operation.

```
ProcessItemsFromStream: PROCEDURE [stream: Stream.Handle] = {
  tH: XToken.Handle ← XToken.HandleFromStream[stream];
  name: XString.ReaderBody ← XToken.Item[tH];
  number: LONG INTEGER;
  UNTIL XString.Empty[@name] DO
    number ← XToken.Decimal[h: tH, signalOnError: FALSE];
    ProcessItem[@name, number]; -- do work
    [] ← XToken.FreeTokenString[@name];
    name ← XToken.Item[tH];
    ENDLOOP;
  [] ← XToken.FreeStreamHandle[tH]};
```

The following example demonstrates how the **XToken** interface could be used to parse input into tokens, optionally followed by switches. In this context, tokens and switches are defined to be any sequence of non-white-space characters, not including the slash character (/).

```
GetToken: PROCEDURE RETURNS [token, switches: XString.ReaderBody] =
  BEGIN
  get: XToken.GetCharProcType = {RETURN[GetCommandLineChar[]]};
```

```
getToken: XToken.Object ← [getChar: get, break: XChar.not];
slash: XChar.Character = '/.ORD;
MyFilter: XToken.FilterProcType = {
  RETURN[SELECT TRUE FROM
    XToken.WhiteSpace[c, data], c = XChar.not = > FALSE,
    c = slash = > FALSE,
    ENDCASE = > TRUE]};
token ← XToken.Filtered[@getToken, NIL, MyFilter];
IF getToken.break = slash THEN switches ← Xoken.Filtered[@getToken, NIL, MyFilter]
ELSE switches ← XString.nullReaderBody;
END;
```

We can extend this example so that the token is defined to be either a sequence of non-white-space characters or a sequence of characters, containing white space characters, between double quotes.

```
GetToken: PROCEDURE RETURNS [token, switches: XString.ReaderBody] =
  BEGIN
  get: XToken.GetCharProcType = {RETURN[GetCommandLineChar[]]};
  getToken: XToken.Object ← [getChar: get, break: XChar.not];
  slash: XChar.Character = '/.ORD;
  doubleQuote: XChar.Character = '".ORD;
  IsQuote: XToken.QuoteProcType = {
    RETURN[IF c = doubleQuote THEN c ELSE XToken.nonQuote]};
  MyFilter: XToken.FilterProcType = {
    RETURN[SELECT TRUE FROM
      XToken.WhiteSpace[c, data], c = XChar.not = > FALSE,
      c = slash = > FALSE,
      ENDCASE = > TRUE]};
  token ← XToken.MaybeQuoted[@getToken, NIL, MyFilter, IsQuote];
  IF getToken.break = slash THEN switches ← Xoken.Filtered[@getToken, NIL, MyFilter]
  ELSE switches ← XString.nullReaderBody;
  END;
```

69.4 Index of Interface Items

II.

APPLICATION INTERFACES

70

ButtonInterchangeDefs

70.1 Overview

The **ButtonInterchangeDefs** interface enables clients to create and enumerate anchored Cusp button frames. Unanchored buttons (those nested within graphic frames) are handled by **GraphicsInterchangeDefs**. The reader should be familiar with **DocInterchangeDefs** and **GraphicsInterchangeDefs** before reading further.

70.1.1 Creating a button

The typical scenario for creating an anchored button in a document is:

```
[doc: doc, ...] ← DocInterchangeDefs.StartCreation[...];
[h, buttonProgram] ← StartButton[
   doc: doc, buttonProps: buttonProps, wantProgramHandle: TRUE];
GraphicsInterchangeDefs.AppendMumbleToButtonProgram[to: buttonProgram, ...];
GraphicsInterchangeDefs.ReleaseButtonProgram[bpPtr: @buttonProgram];
[...] ← DocInterchangeDefs.AppendAnchoredFrame[
   to: doc, type: cuspButton, ..., content: FinishButton[h], ...];
[...] ← DocInterchangeDefs.FinishCreation[docPtr: @doc, ...];
```

70.2 Interface Items

```
StartButton: PROC [
   doc: DocInterchangeDefs.Doc,
   buttonProps: GraphicsInterchangeDefs.ReadonlyButtonProps,
   wantProgramHandle: BOOLEAN ← FALSE]
   RETURNS [
      h: GraphicsInterchangeDefs.Handle,
      buttonProgram: GraphicsInterchangeDefs.ButtonProgram];
```

This interface is used to begin creation of an anchored button (see GraphicsInterchangeDefs.**StartButton** for nested buttons). **doc** must have been obtained from DocInterchangeDefs.**StartCreation** (i.e., **doc** must not be read-only). The **name** inside of **buttonProps** should either be a valid button name or be **NIL**, in which case the button will assume a default name. **wantProgramHandle** determines whether the returned **buttonProgram** is valid or NIL. Pass TRUE for this if you want a non-NIL program for this button. Use the GraphicsInterchangeDefs.Append*ToButtonProgram interfaces to fill in the buttonProgram. If you pass wantProgramHandle=TRUE, you MUST call GraphicsInterchangeDefs.ReleaseButtonProgram (before FinishButton -- see below) on the returned buttonProgram (after you have done all the appending you want). h is a handle you may add graphic objects to. See the GraphicsInterchangeDefs.Add* interfaces. StartButton may raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM, badParameter].

```
FinishButton: PROC [h: GraphicsInterchangeDefs.Handle]
    RETURNS [button: DocInterchangeDefs.Instance];
```

Finishes all the non-program aspects of button creation and returns an instance to pass to the content parameter of DocInterchangeDefs.AppendAnchoredFrame. If you passed wantProgramHandle=TRUE to StartButton, you must call GraphicsInterchangeDefs.ReleaseButtonProgram (and before calling FinishButton).

```
ButtonInfoForAnchoredFrame: PROC [
    doc: DocInterchangeDefs.Doc,
    anchoredFrame: DocInterchangeDefs.Instance,
    props: GraphicsInterchangeDefs.ButtonProps,
    zone: UNCOUNTED ZONE]
    RETURNS [readOnlyButtonProgram: GraphicsInterchangeDefs.ButtonProgram];
```

This interface is used to read the name and program of a button during enumeration. To enumerate the graphic objects contained in the button, call GraphicsInterchangeDefs.Enumerate[..., graphicsContainer: anchoredFrame, ...] (anchoredFrame is a parameter passed to a DocInterchangeDefs.AnchoredFrameProc). props.name is copied into the zone passed in, so the client is responsible for this storage. Use GraphicsInterchangeDefs.EnumerateButtonProgram to enumerate the text in the returned readOnlyButtonProgram. The client must call ReleaseReadOnlyButtonProgram to release the program after enumerating it. Do not call GraphicsInterchangeDefs.ReleaseButtonProgram on the returned readOnlyButtonProgram -- that interface is for releasing client-created button programs.

```
ReleaseReadOnlyButtonProgram: PROC [
    ptr: LONG POINTER TO GraphicsInterchangeDefs.ButtonProgram];
```

This must be called each time ButtonInfoForAnchoredFrame is called. It releases the returned readOnlyButtonProgram. This interface should not be called on programs obtained from either StartButton or GraphicsInterchangeDefs.StartButton -- use GraphicsInterchangeDefs.ReleaseButtonProgram on those.

70.3 Index of Interface Items

71

ChartDataInstallDefs

71.1 Overview

ChartDataInstallDefs provides the ability to install new data in a chart without regard to the type of the chart. Specifics such as line styles or shadings are not affected. Typical clients include those that have changing data depicted in chart form (one such client is the ViewPoint database package).

71.2 Interface Items

The primary type in this interface is the **Handle**, which points to a record of chart information. Clients may obtain a handle by either calling **GetChartFromInstance** or **GetChartFromSelection**. Once the client has a handle, several functions can be performed on the chart; installing new data or validating the chart are some examples.

```
Handle: TYPE = LONG POINTER TO Object;

Object: TYPE = RECORD [
   type: ChartType,
   instance: DocInterchangeDefs.Instance,
   validateChart: ValidateChartProc,
   validateData: ValidateDataProc,
   plot: PlotProc,
   free: FreeProc];

ChartType: TYPE = MACHINE DEPENDENT {bar(0), line(1), pie(2), last(15)};
```

type refers to the manner that the chart displays information. **instance** is a record that has pointers to the chart data. **validateChart** is a call-back procedure to check if the chart can be edited. **validateData** checks the validity of new data to be installed in the chart. **plot** actually installs the data, while **free** releases the handle.

```
ValidateChartProc: TYPE = PROC [h: Handle]
   RETURNS [ChartValidity];

ChartValidity: TYPE = MACHINE DEPENDENT {ok(0), closed(1), readOnly(2), last(15)};
```

ValidateChartProc checks the chart and returns its status. This procedure should be called before any attempt to operate on the chart.

```
ValidateDataProc: TYPE = PROC [
   h: Handle,
   data: Data,
   changes: Selections]
   RETURNS [DataValidity];

DataValidity: TYPE = MACHINE DEPENDENT RECORD [
   v(0): SELECT result(0): DataValidityResult FROM
      ok = > NULL,
      invalidSource = > NULL,
      sizeMismatch = > [extraRows(1): INTEGER, extraCols(2): INTEGER],
      nonNumericValue = > [row(1): CARDINAL, col(2): CARDINAL],
      illegalValue = > [row(1): CARDINAL, col(2): CARDINAL],
      unknown = > NULL,
      last = > NULL,
   ENDCASE];

DataValidityResult: TYPE = MACHINE DEPENDENT {
   ok(0), invalidSource(1), sizeMismatch(2), nonNumericValue(3), illegalValue(4),
   unknown(5), last(15)};
```

ValidateDataProc checks the validity of the new data that the client intends to install. The data is not actually installed in this step. data is a pointer to the current data. changes specifies which items are being validated.

DataValidity indicates the validity of the data and in the case of bad data, some additional information. extraRows is the number of extra rows the new data has relative to the chart's current data table. A negative number means the chart currently has more rows than the data. extraCols is the analagous number for columns. row and col indicate the position of the chart's problem.

```
PlotProc: TYPE = PROC [
   h: Handle,
   data: Data,
   changes: Selections];

Selections: TYPE = PACKED ARRAY Values OF BOOLEAN;

all: Selections = ALL[TRUE];

Values: TYPE = {title, data, rowLabels, colLabels, orientation};
```

PlotProc sets the chart's data and then redraws the chart. data is a pointer to the new data to be installed. changes specifies exactly which data is set.

```
Data: TYPE = LONG POINTER TO DataRec;

DataRec: TYPE = RECORD [
   title: XString.Reader ← NIL,
   data: DataValues,
   rowLabels: Labels ← NIL,
   colLabels: Labels ← NIL,
   orientation: Orientation ← row];

DataValues: TYPE = LONG POINTER TO RowSeq;

RowSeq: TYPE = RECORD [
   rows: SELECT format: DataFormat FROM
      string  => [SEQUENCE nRows: CARDINAL OF StringRow],
      numeric => [SEQUENCE nRows: CARDINAL OF NumericRow]
   ENDCASE];

DataFormat: TYPE = {string, numeric};

Labels: TYPE = LONG POINTER TO LabelSeq;
LabelSeq: TYPE = RECORD [SEQUENCE length:CARDINAL OF XString.Reader];

Orientation: TYPE = {column, row};
```

DataRec contains the data to be installed. title is the title of the chart. data points to a sequence-containing record of data values. rowLabels and colLabels point to sequence-containing records of row and column labels, respectively. orientation specifies whether columns or rows are the chart's data sets.

```
StringRow: TYPE = LONG POINTER TO StringRowElements;
StringRowElements: TYPE = RECORD [SEQUENCE nCols: CARDINAL OF XString.Reader];

NumericRow: TYPE = LONG POINTER TO NumericRowElements;
NumericRowElements: TYPE = RECORD [SEQUENCE nCols: CARDINAL OF XLReal.Number];
```

StringRow points to a sequence-containing record of readers; similarly, NumericRow points to a sequence-containing record of numbers.

Hence, the data to be installed looks like Figure 71.1:

```
GetChartFromInstance: PROC [instance: DocInterchangeDefs.Instance]
   RETURNS [Handle];
```

This procedure returns a handle to the chart given by instance. The result will be NIL if the instance is not a chart. Note that a non-NIL handle doesn't guarantee that the chart is valid; it only guarantees that the instance is a chart. Clients should call the chart's validateChartProc to determine the chart's validity.

```
GetChartFromSelection: PROC RETURNS [Handle];
```

This procedure obtains a handle by converting the current selection. If the current selection is not a chart, NIL will be returned.

```
FreeProc: TYPE = PROC [h: Handle];
```

FreeProc frees the handle passed in by GetChartFromSelection or GetChartFromInstance.

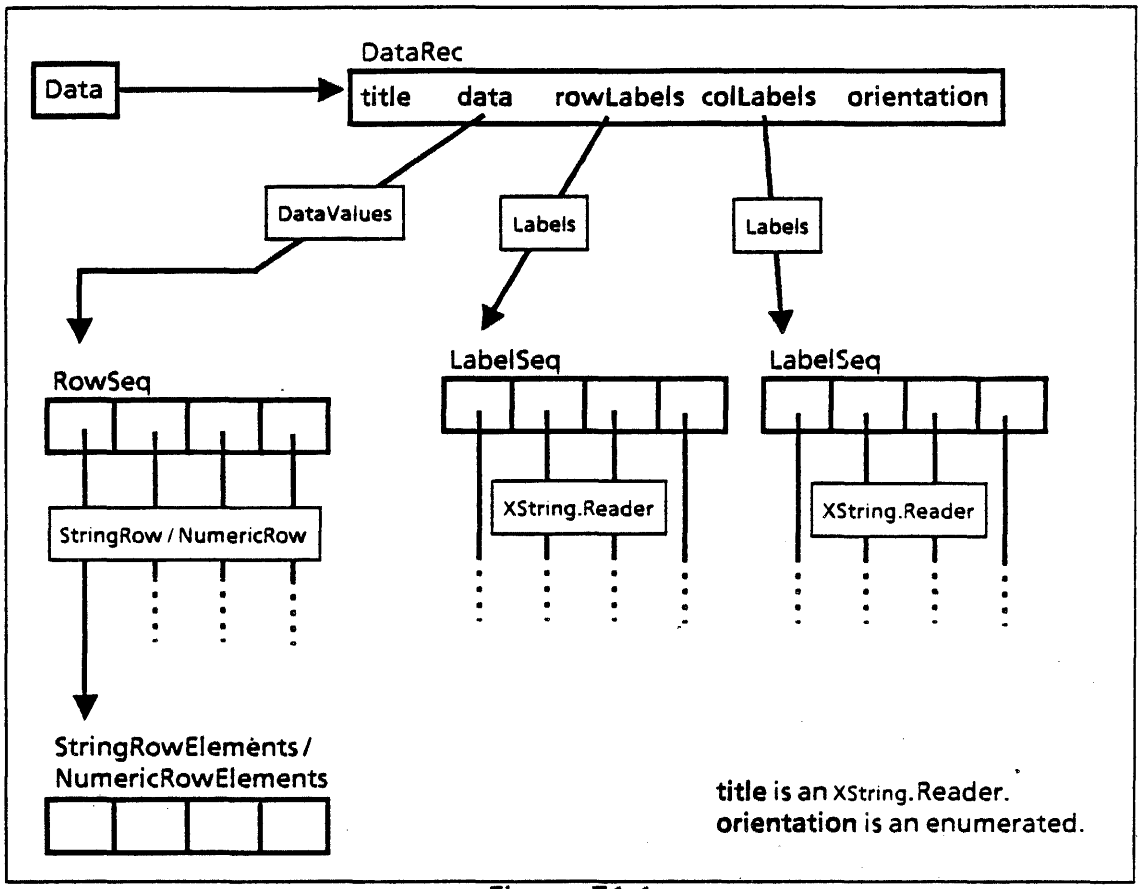

DataRec
Data
title data rowLabels colLabels orientation
DataValues
Labels
Labels
RowSeq
LabelSeq
LabelSeq
StringRow / NumericRow
XString.Reader
XString.Reader
StringRowElements / NumericRowElements
title is an XString.Reader.
orientation is an enumerated.

Figure 71.1

OutOfRoomForGraphics: ERROR;

Raised by **handle.plot** if there is no more room in the document to insert graphic objects (the components of charts).

71.3 Usage

The typical pattern of use for this module is:

```
handle ← GetChartFromSelection[];
IF handle # NIL THEN {
   DO
      <<get raw data;>>
      IF handle.validateChart[handle] # ok THEN {<<error;>> LOOP};
      <<determine which pieces of data are to be changed>>
      <<allocate and fillin data record>>
      IF handle.validateData[handle, data, selections] # ok THEN <<error;>>
      handle.plot[handle, data, selections];
      ENDLOOP;
   handle.free[handle];
   };
```

71.4 Index of Interface Items

72

DocInterchangeDefs

72.1 Overview

The **DocInterchangeDefs** interface enables clients to create a new ViewPoint document or read an existing one. However, it does not support inserting new information or changing or deleting the contents of a document.

DocInterchangeDefs provides procedures to create or read any of the basic document structures, such as text; textual "tiles;" fields; headings and footings; or frames of various types. It does not include procedures to manipulate contents of frames, however.

To create content within frames, the client must use interfaces specific to a particular frame type. The following interfaces are currently available:

GraphicsInterchangeDefs for creating or reading graphics frames

TableInterchangeDefs for creating or reading tables

TextInterchangeDefs for creating or reading text frames

EquationInterchangeDefs for creating or reading equation frames

ButtonInterchangeDefs for creating or reading anchored button frames

These are currently the only frame content interfaces available.

72.1.1 Creating Documents

To create a ViewPoint document, the first step is to call the procedure **StartCreation**. This sets up the data structures for the document and returns a **Doc**, which is a long pointer to an opaque type that represents the document.

The next step is to add information to the document with various **Append*** procedures: **AppendAnchoredFrame**, **AppendAnchoredFrameX**, **AppendBreak**, **AppendChar**, **AppendColumnBreak**, **AppendField**, **AppendNewParagraph**, **AppendPageBreak**, **AppendPFC** (Page Format Character), **AppendText**, or **AppendTile**.

With **AppendAnchoredFrame**, the client would typically call an operation in an interface such as **GraphicsInterchangeDefs** or **TableInterchangeDefs** to create the contents of the frame, and then call **AppendAnchoredFrame** to add that frame and its contents to the document. With **TextInterchangeDefs**, the client calls **AppendAnchoredFrame** first and then uses **TextInterchangeDefs** to append information to the text frame.

AppendField, AppendPFC, and **AppendTile** all have return values: this allows the client to call **Append*** routines recursively to add text and formatting information to fields, tiles, or PFC headers.

When the document contains all the desired information, the client should call **FinishCreation**, which returns an **NSFile.Handle** for the newly created file.

72.1.2 Enumerating documents

To enumerate the contents of an existing ViewPoint document, the client should start by calling **Open**, which opens the document and returns a **Doc** handle for that document. The next step is to call **Enumerate**, passing in the **Doc** and an **EnumProcs** record. The **EnumProcs** record contains a set of callback procedures, one for each of the following structures: anchored frame, column break, field, new paragraph, page break, page format character, text, tile, or break character (this last via **spare1Proc**).

Enumerate proceeds sequentially from the beginning of the document: as it comes to different structures within the document, it calls the appropriate callback procedure, which handles it appropriately. Each of these procedures returns a boolean value **stop**; if any one of the procedures returns **stop = TRUE**, the enumeration will terminate. If **stop** is never **TRUE**, the enumeration will continue to the end of the document.

Note that the enumeration proceeds according to the "main flow" of text within a document--the text sequence that contains page format characters and frame anchor characters. This means that an **AnchoredFrameProc** will be called not when the frame itself is reached, but rather when the frame's anchor character is reached. **DocInterchangeDefs** knows nothing about how the document has been arranged structurally by operations such as pagination. As a consequence of this, the enumeration can never know what page it is on.

When the enumeration is complete, the client should call **Close** to free all associated data structures and close any open file handles to the document.

Note that Creation and Enumeration are totally separate activities and procedures/handles associated with one should not be used with the other. Enumeration is a readonly operation; no editing should be attempted while it is in progress (the results are undefined). Likewise, Enumeration should not be attempted during Creation.

72.2 Interface Items

72.2.1 Data types

The basic data structure of **DocInterchangeDefs** is the **TextContainer**, which is any object that can contain text. A **TextContainer** can be a caption, document, field, heading, footing, or spare (spares are for future compatability).

```
TextContainer: TYPE = RECORD [
    var: SELECT type: * FROM
```

```
        caption => [h: Caption],
        doc => [h: Doc],
        field => [h: Field],
        heading => [h: Heading],
        footing => [h: Footing],
        spare1 => [h: SpareTC],
        spare2 => [h: SpareTC],
        spare3 => [h: SpareTC],
        spare4 => [h: SpareTC],
    ENDCASE];

Caption: TYPE = LONG POINTER TO CaptionObject;
CaptionObject: TYPE;

Doc: TYPE = LONG POINTER TO DocObject;
DocObject: TYPE;

Field: TYPE = LONG POINTER TO FieldObject;
FieldObject: TYPE;

Heading: TYPE = LONG POINTER TO HeadingObject;
HeadingObject: TYPE;

Footing: TYPE = LONG POINTER TO FootingObject;
FootingObject: TYPE;

Tile: TYPE = LONG POINTER TO TileObject;
TileObject: TYPE;

SpareTC: TYPE = LONG UNSPECIFIED;

Instance: TYPE[2];
instanceNil: Instance = LOOPHOLE[LONG[0]];
```

Note that **TextContainers** must contain at least one **newParagraph** character, since the paragraph properties of any text are always inherited from the preceding **newParagraph** character. The **DocInterchange** implementation supplies the initial **newParagraph** characters as required; the client may assume they already exist. (The client is free to append them anyway. The implementation ensures that if the client appends one at the start of the **TextContainer**, two won't appear. The client's paragraph and font properties on the **newParagraph** they appended will have precedence.)

An **Instance** is a handle to one of a certain class of objects within a document. Many, though not all, objects within a document can be uniquely identified and accessed via an **Instance**. In general, instances form the bridge between **DocInterchangeDefs** and the frame-content specific Interchange interfaces such as **GraphicsInterchangeDefs** and **TableInterchangeDefs**: **DocInterchangeDefs** will provide an instance which may be passed to operations in other Interchange interfaces. No object in any document may be accessed through **instanceNil**.

72.2.2 Creating documents

72.2.2.1 Initializing a document

The client calls **StartCreation** to initiate the document creation process.

```
StartCreation: PROC [
    paginateOption: PaginateOption ← compress,
    wantHeadingHandles, wantFootingHandles: BOOL ← FALSE,
    initialFontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    initialParaProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    initialPageProps: DocInterchangePropsDefs.ReadonlyPageProps ← NIL]
    RETURNS [
        doc: Doc,
        leftHeading, rightHeading: Heading,
        leftFooting, rightFooting: Footing,
        status: StartCreationStatus];

PaginateOption: TYPE = MACHINE DEPENDENT {
    none(0), simple, compress, firstAvailable, lastAvailable(255)};
```

paginateOption specifies the type of pagination that will occur when the client calls **FinishCreation**. It is specified here rather than in **FinishCreation** to enable performance optimizations based on the type of pagination that will eventually occur.

> **compress** pagination provides all the outward signs of pagination, such as page format properties, and leaves the structure of the document in its optimized (most compact) form.
>
> **simple** pagination provides the outward signs of pagination but does not leave the document in its optimized form, so subsequent editing may be slower than with **compress** pagination. **simple** pagination is somewhat faster than **compress**.
>
> **none** leaves the document in its raw form. This can lead to very slow editing, and potentially to loss of data. If the document will be more than a few pages long, client must specify a **paginateOption** other than **none** to avoid losing data.

wantHeadingHandles and **wantFootingHandles** specify whether the first page format character in the document will have headings and footings.

initialFontProcs, **initialParaProcs**, and **initialPageProps** specify the initial properties for the document. If you do not specify a field of initial properties, **StartCreation** will use the document default properties. (For information on document default properties, see the DocInterchangePropsDefs chapter)

In the **pageProps**, the client must ensure that the page margins leave at least one inch (72 points) for text. That is, (left margin + right margin + 72 < = page width), and (top margin + bottom margin + 72 < = page height).

StartCreation returns a **Doc** handle, handles for headings and footings, and a status. The **Doc** handle represents the new document. The client should pass this handle to the

Append* procedures described below to add information to the document, and then eventually release the handle with a call to **FinishCreation**.

If the client releases the handle without ever calling any **Append*** routines, the file will contain a 1-page document containing a single **newParagraph** and **pageFormat** character, with the initial font, paragraph, and page props as specified.

The heading and footing handles that are returned will be NIL unless the client specified **wantHeadingHandles** or **wantFootingHandles** = TRUE. If the headings or footings are valid, the client should call various **Append*** routines to add text and formatting information, and then later release each handle with a call to **ReleaseHeading** or **ReleaseFooting**. See section 72.2.2.3 for details.

StartCreation (and all subsequent editing operations on the **Doc**) may be called either normally or by a forked process running in the background. Note that while background creation is allowed, the client may have only one process accessing a particular document at a time. If forked Creation is desired, the client must call **StartCreation** in a process that is running at **Process.priorityBackground**. **StartCreation** must do special things to be able to run in a forked process, and it detects this situation by examining the calling process' priority. The client may change the priority once **StartCreation** returns, if desired. All creation operations execute normally in a forked process, although the client must make a special call following **FinishCreation**. See section 72.2.2.4.

```
StartCreationStatus: TYPE = MACHINE DEPENDENT {
   ok(0), notEnoughDiskSpace, notEnoughVM, firstAvailable, lastAvailable(255)};
```

StartCreation returns a status code, which can have any of the following values:

ok	Everything was fine.
notEnoughDiskSpace	There isn't enough disk space to perform the operation.
notEnoughVM	There isn't enough contiguous virtual memory to create.

72.2.2.2 Adding to a document

The **Append*** routines below add various kinds of information to **TextContainers**.

AppendChar, AppendField, AppendNewParagraph, AppendText, and **AppendTile** take a **TextContainer** as a parameter and add the specified information to that container. The remaining procedures (**AppendAnchoredFrame, AppendAnchoredFrameX, AppendBreak, AppendColumnBreak, AppendPFC, AppendPageBreak**) take only a **Doc**, and not a general purpose **TextContainer**; other **TextContainers** cannot contain the various special characters.

With all of these procedures, the client must manage the storage for the property records or other data structures passed in, except for handles obtained from the interface itself. The storage for the properties must remain valid during the call to **Append***; after **Append*** returns, the client may do anything it chooses with the storage (typically, free it).

The **Append*** procedures often allow the client to set font, paragraph, or page properties. Defaulting any of these arguments will cause the newly appended text or object to inherit the properties of the preceding text/object and not the application-wide default properties.

If an **Append*** routine returns a non-NIL handle, the client is responsible for later freeing that handle with a call to an appropriate **Release*** routine. See section 72.2.2.3 for details.

```
AppendAnchoredFrame: PROC [
    to: Doc,
    type: AnchoredFrameType,
    anchoredFrameProps: DocInterchangePropsDefs.ReadonlyFrameProps,
    content: Instance ← instanceNil,
    wantTopCaptionHandle, wantBottomCaptionHandle,
    wantLeftCaptionHandle, wantRightCaptionHandle: BOOL ← FALSE,
    anchorFontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [
        anchoredFrame: Instance,
        topCaption, bottomCaption,
        leftCaption, rightCaption: Caption];
```

AppendAnchoredFrame appends the anchored frame **type** with properties **anchoredFrameProps** to the document **Doc**.

```
AnchoredFrameType: TYPE = MACHINE DEPENDENT {
    nil(0), bitmap, cuspButton, equation, graphics, IMG, table, text,
    illustrator, firstAvailable, lastAvailable(255)};
```

content is the contents of the frame. Currently, there are interfaces to support creating graphics, table, text, equation, and button frames.

want*CaptionHandle specifies which captions the frame should have. **anchorFontProps** specifies the font properties of the frame anchor. Changing the font properties of the anchor does not affect how that anchor appears on the display, but does affect the default properties that succeeding characters will inherit.

AppendAnchoredFrame returns handles to the frame and its captions. The caption handles will be non-NIL only if the client specified TRUE for the corresponding **want*CaptionHandle** parameter. The client must later release each valid caption handle with **ReleaseCaption**.

The return parameter **anchoredFrame** is currently used only by the **TextInterchangeDefs** interface for appending text frames.

```
AppendAnchoredFrameX: PROC [
    to: Doc,
    type: AnchoredFrameType,
    anchoredFrameProps: DocInterchangePropsDefs.ReadonlyFrameProps,
    content: Instance ← instanceNil,
    wantTopCaptionHandle, wantBottomCaptionHandle,
    wantLeftCaptionHandle, wantRightCaptionHandle: BOOL ← FALSE,
    anchorFontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [
        anchoredFrame: Instance,
        topCaption, bottomCaption,
        leftCaption, rightCaption: Caption];
```

This operation is similar to **AppendAnchoredFrame**, except table frames are handled a bit differently. For these, whatever frame size is specified in the **anchoredFrameProps** will be used without modification. The normal **AppendAnchoredFrame** ignores the frame size passed in for tables and always creates the a table frame that just fits around the enclosed table. (In the next major release the original **AppendAnchoredFrame** should function as this does, and the extra interface will go away.) This operation is currently defined in **DocInterchangeExtra3Defs**.

```
BreakProps: TYPE = LONG POINTER TO BreakPropsRecord;
ReadonlyBreakProps: TYPE = LONG POINTER TO READONLY BreakPropsRecord;
BreakPropsRecord: TYPE = RECORD [
    breakType: BreakType,
    spare1: LONG CARDINAL];

BreakType: TYPE = MACHINE DEPENDENT {
    newPage(0), newLeftPage, newRightPage, newColumn, firstAvailable,
    lastAvailable(255)};

AppendBreak: PROC [
    to: TextContainer,
    breakProps: ReadonlyBreakProps,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

AppendBreak appends a break character to the document. **fontProps** are the properties of the break character; these properties do not affect the appearance of the character itself, but they do affect the properties that succeeding characters will inherit. This operation and its associated types are currently defined in **DocInterchangeExtra1Defs**. **AppendBreak** replaces **AppendColumnBreak** and **AppendPageBreak**, which are obsolete (though still supported).

```
AppendChar: PROC [
    to: TextContainer,
    char: XChar.Character,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];
```

AppendChar appends one or more copies of the text character **char** to the specified **TextContainer**. **nToAppend** specifies the number of copies of the character that are to be appended; **fontProps** specifies the character properties.

```
AppendColumnBreak: PROC [
    to: Doc, fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

AppendColumnBreak appends a column break character to a document. **fontProps** are the properties of the column break character; these properties do not affect the appearance of the character itself, but they do affect the properties that succeeding characters will inherit. Note: this operation is obsolete, but still supported. We recommend using **AppendBreak** instead.

```
AppendField: PROC [
    to: TextContainer,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [field: Field];
```

AppendField appends a field to the specified **TextContainer**. **AppendField** returns a field: the client can then add information to the field by using the **Field** as the **TextContainer** in other calls to **Append*** routines (but not **AppendField** again.) When the client is through with the field, it must release it via **ReleaseField**. See section 72.2.2.3.

Note that the client cannot set the fill-in order of the fields when they are appended to the document. This may be done via **AppendItemToFillInOrder**, which is described in section 72.2.5.

```
AppendNewParagraph: PROC [
    to: TextContainer,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];
```

AppendNewParagraph appends one or more new paragraph characters to a **TextContainer** object. **nToAppend** specifies the number of characters to be appended. **paraProps** and **fontProps** specify the properties for the paragraph. If **paraProps** is NIL, the new paragraph inherits the props of the previous paragraph; otherwise, **paraProps** determines the properties of the paragraph.

Note that **TextContainers** always contain at least one **newParagraph** character. The client does not have to provides these initial **newParagraph** characters; **DocInterchange** implementation supplies them as required. (see the end of section 72.2.1)

```
AppendPageBreak: PROC [
    to: Doc, fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

AppendPageBreak appends a page break character to the text of a document. The **fontProps** do not affect the appearance of the page break character itself, but they do affect the properties that succeeding characters will inherit. Note: this operation is obsolete, but still supported. We recommend using **AppendBreak** instead.

```
AppendPFC: PROC [
    to: Doc,
    pageProps: DocInterchangePropsDefs.ReadonlyPageProps,
    wantHeadingHandles, wantFootingHandles: BOOL ← FALSE,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [
        leftHeading, rightHeading: Heading,
        leftFooting, rightFooting: Footing];
```

AppendPFC appends a page format character to the main document text.

pageProps specify the properties for the new page. The client must ensure that the page margins leave at least one inch (72 points) for text. That is, (**left margin + right margin + 72 <= page width**), and (**top margin + bottom margin + 72 <= page height**).

The heading and footing handles that are returned will be NIL unless the client specified **wantHeadingHandles** or **wantFootingHandles = TRUE**.

If the heading and footing handles are valid, the client can then use them as **TextContainers** for further calls with **Append*** procedures. If the headers are to be the same on left and right pages, only **leftHeading** need contain the heading; **rightHeading** should be NIL. The same rule applies for **leftFooting** and **rightFooting**.

When creating a heading or footing, the client should note that there are no automatic positioning parameters for information in headers and footers; the client must call the appropriate **Append*** procedures to add the desired text and position it with standard text formatting, such as white-space characters, paragraph alignment, leading, line height, and tabs.

Additionally, there is no page number pattern; the client must place any surrounding text directly in the heading/footing text, inserting the # character at the position(s) where a page number is desired. (Note that there is a procedure, **DocInterchangePropsDefs.GetPageNumberDelimiter**, that returns this character.)

The client must later free every non-NIL heading or footing with a call to **ReleaseHeading** or **ReleaseFooting**.

```
AppendText: PROC [
    to: TextContainer,
    text: XString.Reader,
    textEndContext: XString.Context,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

AppendText appends the **text** with the specified properties to the **TextContainer**. For efficiency, the client should pass the appropriate **textEndContext** if it is known (just like **XString.AppendReader**). **text** may not contain **newParagraph** characters ([set: 0, code: 35B]). Use **AppendNewParagraph** to append these.

```
AppendTile: PROC [
    to: TextContainer,
    type: Atom.ATOM,
    data: LONG POINTER ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [tile: Tile];
```

AppendTile is for future use. The tile type and data format are defined elsewhere, agreed upon by parties on either side of this interface.

72.2.2.3 Releasing storage

```
ReleaseCaption: PROC [captionPtr: LONG POINTER TO Caption];

ReleaseField: PROC [fieldPtr: LONG POINTER TO Field];

ReleaseHeading: PROC [headingPtr: LONG POINTER TO Heading];

ReleaseFooting: PROC [footingPtr: LONG POINTER TO Footing];

ReleaseTile: PROC [tilePtr: LONG POINTER TO Tile];

ReleaseSpare1: PROC [ptr: LONG POINTER TO SpareTC];

ReleaseSpare2: PROC [ptr: LONG POINTER TO SpareTC];

ReleaseSpare3: PROC [ptr: LONG POINTER TO SpareTC];

ReleaseSpare4: PROC [ptr: LONG POINTER TO SpareTC];
```

The client must call **ReleaseCaption**, **ReleaseField**, **ReleaseFooting**, **ReleaseHeading**, **ReleaseTile**. or **ReleaseSpare** to release resources associated with any non-NIL handle obtained from any **Append*** procedure.

After calling **Release***, the handle will be invalid. To help prevent use of an invalid handle, the **Release*** routines take a pointer to the handle, and set the handle itself to NIL. (This is similar to Mesa's FREE operation.)

72.2.2.4 Finalizing a document

```
FinishCreation: PROC [
    docPtr: LONG POINTER TO Doc,
    checkAbortProc: CheckAbortProc ← NIL,
    checkAbortClientData: LONG POINTER ← NIL]
    RETURNS [
        docFile: NSFile.Handle,
        session: NSFile.Session,
        status: FinishCreationStatus];
```

When the document is complete, the client must call **FinishCreation** to finalize the document and release the **Doc** handle. **FinishCreation** returns an **NSFile.Handle** to the newly-created document, an **NSFile.Session**, and a status. The file handle is valid in the returned NSFile session. The **session** returned will be the default session if the client called **FinishCreation** normally (not by a forked background process). If **StartCreation** was called by a forked background process, then the returned session will be a private session created by **StartCreation**. The client must kill this private session by calling **NSFile.Logoff[session]** after it's finished processing the document file. The document that **FinishCreation** provides will be in paginated form if the client so specified in **StartCreation**.

```
CheckAbortProc: TYPE = PROC [clientData: LONG POINTER] RETURNS [abort: BOOL];
```

If the client specified a **checkAbortProc**, then a call-back procedure will be invoked before the call to **FinishCreation** returns; this call-back can abort the document's completion. **checkAbortClientData** is a client defined argument and is passed into the call-back procedure.

```
FinishCreationStatus: TYPE = MACHINE DEPENDENT {ok(0),
    aborted, okButNotEnoughDiskSpaceToPaginate, okBuNotEnoughVMToPaginate,
    okButUnknownPaginateProblem, firstAvailable, lastAvailable(255)};
```

FinishCreation also returns a status code, which can have any of the following values:

aborted	do not complete the document.
okButNotEnoughDiskSpaceToPaginate, **okBuNotEnoughVMToPaginate,** **okButUnknownPaginateProblem**	the document is finished but left unpaginated

This document file is temporary, and will be purged from the NSFile system if a reboot occurs before it is made permanent. To make the file permanent, the client should call move it to the current user desktop with **NSFile.Move**, followed by a call to **StarDesktop.AddReference** to put the icon on the display. (See section 72.3 for an example of this.)

```
AbortCreation: PROC [docPtr: LONG POINTER TO Doc];
```

AbortCreation aborts document creation and deallocates the storage associated with that document. **AbortCreation** will kill the private session if **StartCreation** was called by a background process, so the client should not call **NSFile.Logoff[session]** after having called **AbortCreation**.

72.2.2.5 Utilities

The following procedures are utilities that may be of use to the client creating a document.

```
GetModeProps: PROC [doc: Doc, modeProps: DocInterchangePropsDefs.ModeProps];

SetModeProps: PROC [
    doc: Doc,
    modeProps: DocInterchangePropsDefs.ReadonlyModeProps,
    selections: DocInterchangePropsDefs.ModeSelections];
```

Get or set the mode properties for the document; these procedures may be called at any time. When setting mode properties, only those properties designated by TRUE selections will be changed.

```
SetCurrentParagraphProps: PROC [
    textContainer: TextContainer,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps];
```

SetCurrentParagraphProps can be called at any time, such as in the middle of a paragraph or (even if it makes no sense) repeatedly with different properties. If it is called repeatedly in the same paragraph, only the most recent call will remain in effect. The client can call this procedure on any **TextContainer**, including a document.

SetCurrentParagraphProps affects the entire current paragraph, including any portion not yet appended at the time it is called. The properties also affect all subsequent paragraphs unless the client overrides the properties with new ones passed to **AppendNewParagraph**, or by another call to **SetCurrentParagraphProps**.

Note, however, that setting paragraph properties on a **TextContainer** will cause an error if that **TextContainer** does not contain any paragraph characters. Although **DocInterchange** does guarantee that every **TextContainer** will contain at least one new paragraph character, those paragraph characters are added (if necessary) during the **Append*** routines. Thus, calling **SetCurrentParagraphProps** before calling any **Append*** routines will cause an error. To avoid this problem, the client can simply call **AddNewParagraph** to ensure that the **TextContainer** does have a paragraph character. Since the **Append*** routines only add a new paragraph if necessary, this will not cause duplication.

72.2.3 Enumerating documents

72.2.3.1 Open

```
Open: PROC [
    docFileRef: NSFile.Reference,
    session: NSFile.Session ← NSFile.nullSession,
    password: XString.Reader ← NIL]
    RETURNS [doc: Doc, status: OpenStatus];
```

To enumerate a document, the first step is to call **Open**. **Open** takes a **NSFile.Reference** for a file and opens it for reading in the NSFiling session specified by the **session** argument. The client should then pass **Doc** returned to **Enumerate**, which will parse the document. **Open** (and all subsequent reading operations on the **Doc**) may be called either normally or by a forked process running in the background. Note that while background enumeration is allowed, the client may have only one process accessing a particular document at a time. All reading operations execute normally in a forked process. If **session** is defaulted, **NSFile.GetDefaultSession[]** will be used.

password is provided in anticipation of future password-locking of documents. **password** is currently ignored.

```
OpenStatus: TYPE = MACHINE DEPENDENT {
    ok(0), malFormed, incompatible, notLocal, outOfDiskSpace, outOfVM, busy,
    invalidPassword, firstAvailable, lastAvailable(255)};
```

Open also returns a status code, which can have any of the following values:

ok	Everything was fine
malFormed	The Document is inconsistent internally.
incompatible	The document is of a version that the VP Document Editor cannot open.

notLocal	The document is not on the workstation, so it cannot be opened.
outOfDiskSpace	There isn't enough disk space to open the document.
outOfVM	There isn't enough contiguous virtual memory.
busy	Another process is using the file (e.g. background pagination).
invalidPassword	The user does not have the credentials to open the document.

72.2.3.2 Enumerate

```
Enumerate: PROC [
    textContainer: TextContainer,
    procs: EnumProcs,
    clientData: LONG POINTER ← NIL]
    RETURNS [dataSkipped: BOOL];
```

Enumerate parses the contents of the specified **TextContainer**. The client may pass a NIL value as the **textContainer**--that call to **Enumerate** will do nothing. (NIL **textContainers** may be passed to the client; the client need not check for NIL before attempting to enumerate one. This does not imply, however that non-NIL **textContainers** have any content--they may be empty yet have a non-NIL handle.)

procs is a record that contains client-defined callback procedures to enumerate the various kinds of structures that can be found in a **TextContainer**.

dataSkipped will be TRUE if **Enumerate** encountered an object that it didn't recognize, or if it encountered an object for which a client call-back procedure was not supplied.

```
EnumProcs: TYPE = LONG POINTER TO EnumProcsRecord;
```

```
EnumProcsRecord: TYPE = RECORD [
    anchoredFrameProc: AnchoredFrameProc ← NIL,
    columnBreakProc: ColumnBreakProc ← NIL,
    fieldProc: FieldProc ← NIL,
    newParagraphProc: NewParagraphProc ← NIL,
    pageBreakProc: PageBreakProc ← NIL,
    pfcProc: PFCProc ← NIL,
    textProc: TextProc ← NIL,
    tileProc: TileProc ← NIL,
    spare1Proc: SpareProc ← NIL,
    spare2Proc: SpareProc ← NIL,
    spare3Proc: SpareProc ← NIL,
    spare4Proc: SpareProc ← NIL];
```

Each of the procedures in an **EnumProcsRecord** takes as parameters the properties of the structure and its content when appropriate. Note that the storage for the properties passed to these procedures is temporary; the client must explicitly copy any properties it wishes to save. For a description of the various properties, see the corresponding **Append*** routines.

spare1Proc, if not NIL, will be called for all break characters. Its **data** parameter should be LOOPHOLEd into a **ReadonlyBreakPropsForEnum** (currently defined in **DocInterchangeExtra1Defs**). **columnBreakProc** and **pageBreakProc** are obsolete, but still supported. If **spare1Proc** is not NIL, then both **columnBreakProc** and **pageBreakProc** must be. Conversely, if either columnBreakProc or pageBreakProc is not NIL, then spare1Proc must be. We recommend using spare1Proc for enumerating break characters, as it is more robust.

```
AnchoredFrameProc: TYPE = PROC[
    clientData: LONG POINTER,
    type: AnchoredFrameType,
    anchorFontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    anchoredFrame: Instance,
    anchoredFrameProps: DocInterchangePropsDefs.ReadonlyFrameProps,
    content: Instance,
    topCaption,
    bottomCaption,
    leftCaption,
    rightCaption: Caption]
    RETURNS [stop: BOOL ← FALSE];

ColumnBreakProc: TYPE = PROC[
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps]
    RETURNS [stop: BOOL ← FALSE];

FieldProc: TYPE = PROC[
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    field: Field]
    RETURNS [stop: BOOL ← FALSE];

NewParagraphProc: TYPE = PROC[
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps]
    RETURNS [stop: BOOL ← FALSE];

PageBreakProc: TYPE = PROC[
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps]
    RETURNS [stop: BOOL ← FALSE];

PFCProc: TYPE = PROC[
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    pageProps: DocInterchangePropsDefs.ReadonlyPageProps,
    leftHeading, rightHeading: Heading,
    leftFooting, rightFooting: Footing]
    RETURNS [stop: BOOL ← FALSE];
```

In a PFCProc, if the headers are the same on left and right pages, only leftHeading will contain the heading; rightHeading will be NIL. (Of course, leftHeading can be NIL if it has no content.) The same rule applies for leftFooting and rightFooting.

```
TextProc: TYPE = PROC [
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    text: XString.Reader,
    textEndContext: XString.Context]
    RETURNS [stop: BOOL ← FALSE];
```

In a **TextProc**, **textEndContext** will always be accurate; it will never be XString.unknownContext.

```
TileProc: TYPE = PROC [
    clientData: LONG POINTER,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    type: Atom.ATOM,
    data: LONG POINTER,
    tile: Tile]
    RETURNS [stop: BOOL ← FALSE];
```

```
SpareProc: TYPE = PROC [
    clientData: LONG POINTER,
    data: LONG UNSPECIFIED]
    RETURNS [stop: BOOL ← FALSE];
```

As it encounters an object of a particular type, **Enumerate** will call the appropriate procedure. If the client defaults a particular procedure, **Enumerate** will ignore any objects of that type.

Each procedure has a **stop** return parameter; the enumeration will stop if **stop** ever has the value TRUE. Some of the procedures also have a **TextContainer** handle as a parameter; the client can use this **TextContainer** recursively in other calls to **Enumerate** to obtain the contents of the **TextContainer**.

The **clientData** pointer passed in to Enumerate is passed to the callback procedures invoked by (that call to) Enumerate. (The **clientData** may be different at different recursion levels, of course.)

The handle (header, caption, etc.) supplied to the client in the call-back is readonly and is valid only during the call-back's invocation; the client is not reponsible for releasing this handle. It is possible for such a handle to be NIL; a NIL handle means that the corresponding object has no text content.

Note that the enumeration does include the initial paragraph (and possible page format characters) that every **TextContainer** has. Thus, when copying a document into a new document, the client should be careful to avoid copying the initial paragraph and page format characters and then appending some more initial ones, since that would cause duplication.

72.2.3.3 Close

```
Close: PROC [docPtr: LONG POINTER TO Doc];
```

When through with an enumeration, the client should call **Close**, which releases storage associated with the **Doc** handle and sets the **Doc** handle to NIL.

72.2.4 Errors

```
Error: ERROR [why: ErrorCode];

ErrorCode: TYPE = MACHINE DEPENDENT {
    containerFull(0), documentFull, readonlyDoc, outOfDiskSpace, outOfVM,
    objectIllegalInContainer, badParameter, unimplemented, firstAvailable,
    lastAvailable(255)};
```

Any of the Append* procedures can raise an error, which can be one of the following types:

containerFull	there is no more room to append to this container.
documentFull	no more room in the document.
readonlyDoc	document opened in ReadOnly mode.
outOfDiskSpace	not enough disk space for the operation.
outOfVM	not enough virtual memory for the operation.
objectIllegalInContainer	attempted to add an object to a container that does not support that object type.
badParameter	one of the arguments specified was invalid in this context.
unimplemented	this function is not supported.

Do not call any Interchange operations from within a catch phrase of **Error**.

72.2.5 Fill-in Order

DocInterchangeDefs provides procedures to append, enumerate, and clear the fill-in order of fields and tables.

```
AppendItemToFillInOrder: PROC [
    doc: Doc,
    fillInOrderItemName: XString.Reader,
    itemType: FillInOrderItemType];
```

doc is the document that contains the field or table.

fillInOrderItemName is the name of the object being added to the fill-in order.

```
FillInOrderItemType: TYPE = MACHINE DEPENDENT {
    field(0), table, firstAvailable, lastAvailable(255)};
```

FillInOrderItemType specifies the type of object that will be added to the fill-in order.

```
EnumerateFillInOrder: PROC [
    doc: Doc,
    proc: FillInOrderProc,
    clientData: LONG POINTER ← NIL];

FillInOrderProc: TYPE = PROC [
    clientData: LONG POINTER,
    fillInOrderItemName: XString.Reader,
    itemType: FillInOrderItemType]
    RETURNS [stop: BOOL ← FALSE];
```

proc is a call-back procedure that is invoked once for each object in the fill-in order. The arguments passed into proc include the name of the enumerated object as well as its type. The **FillInorderProc** can return stop = TRUE to halt the enumeration.

clientData is client defined data that is passed to **proc**.

```
ClearFillInOrder: PROC [doc: Doc];
```

Clear the fill-in order for the entire document.

72.3 Usage/Examples

Here is an example of both enumeration and creation. This program adds two commands to the Attention Window: "Copy Most of Doc (Forked)" and "Copy Most of Doc (Notifier)". When called, these commands check to see if the current selection is a document. If it is, then the program enumerates the contents of that document and copies the information into a new document.

```
DIRECTORY
  ...;

DocExample: PROGRAM IMPORTS ..., DocInterchangeDefs, ... = {

  -- Types

  DICtxtHandle: TYPE = LONG POINTER TO DICtxt;
  DICtxt: TYPE = RECORD [
    sourceDoc, targetDoc: DocInterchangeDefs.Doc,
    ignoreNewPar, ignorePFC, aborted, error: BOOLEAN];
  << A DICtxtHandle is passed as clientData to procs called by
     DocInterchangeDefs.Enumerate. >>

  << The following types are used to hold copied heading or footing text. >>
  HeadFootText: TYPE = LONG POINTER TO HeadFootTextRec;
  HeadFootTextRec: TYPE = RECORD [
    length: CARDINAL, list: SEQUENCE maxLength: CARDINAL OF ChunkRec];
  ChunkRec: TYPE = RECORD [
    fontProps: DocInterchangePropsDefs.FontPropsRecord,
    v: SELECT type: * FROM
      np => [paraProps: DocInterchangePropsDefs.ParaPropsRecord],
      text => [rb: XString.ReaderBody, textEndContext: XString.Context],
      ENDCASE];

  -- Constants
```

```
z: UNCOUNTED ZONE = BWSZone.shortLifetime;

-- Variables

diEnumProcsRec: DocInterchangeDefs.EnumProcsRecord ← [
   anchoredFrameProc: AppendAnchoredFrameToTargetDoc,
   fieldProc: AppendFieldToTargetDoc,
   newParagraphProc: AppendNewParToTargetDoc, pfcProc: AppendPFCToTargetDoc,
   textProc: AppendTextToTargetDoc, spare1Proc: AppendBreakToTargetDoc];
diEnumProcs: DocInterchangeDefs.EnumProcs = @diEnumProcsRec;

-- Copy contents of current selection to new doc.
MakeDoc: PROC [docFileRef: NSFile.Reference, background: BOOL] = {
   nameRB: XString.ReaderBody ←
      XString.FromSTRING["Copying Most of Doc"L, TRUE];

   CallBack: BackgroundProcess.CallBackProc = {
      sourceDoc: DocInterchangeDefs.Doc;
      openStatus: DocInterchangeDefs.OpenStatus;
      « Use a private session when enumerating source doc so that user can't select
         and open the document while we're reading it. »
      sourceDocSession: NSFile.Session =
         (IF background THEN NSFile.Logon[
          Atom.GetProp[
          Atom.MakeAtom["CurrentUser"L],
         Atom.MakeAtom["IdentityHandle"L]] ↑.value]
          ELSE NSFile.GetDefaultSession[]);
      finalStatus ← quietSuccess;
      [sourceDoc, openStatus] ←
         DocInterchangeDefs.Open[docFileRef, sourceDocSession];
      IF openStatus = ok THEN { -- source ok, attempt to copy contents.
         targetDoc: DocInterchangeDefs.Doc;
         diCtxt: DICtxt;
         docFile: NSFile.Handle;
         targetDocSession: NSFile.Session;
         fontProps: DocInterchangePropsDefs.FontPropsRecord;
         paraProps: DocInterchangePropsDefs.ParaPropsRecord;
         pageProps: DocInterchangePropsDefs.PagePropsRecord;
         sourceLeftHeading, sourceRightHeading,
         sourceLeftFooting, sourceRightFooting: HeadFootText;
         targetLeftHeading, targetRightHeading: DocInterchangeDefs.Heading;
         targetLeftFooting, targetRightFooting: DocInterchangeDefs.Footing;

         CheckAbort: DocInterchangeDefs.CheckAbortProc = {
            abort ← diCtxt.aborted ← BackgroundProcess.UserAbort[];
            IF abort THEN BackgroundProcess.ResetUserAbort[];
            };

         -- start of "openStatus = ok" code
         {
         GetInitialDocProps[
            docFileRef, sourceDoc, sourceDocSession, @fontProps, @paraProps,
            @pageProps, @sourceLeftHeading, @sourceRightHeading,
            @sourceLeftFooting, @sourceRightFooting, z];
         [targetDoc, targetLeftHeading, targetRightHeading,
            targetLeftFooting, targetRightFooting, ] ←
            DocInterchangeDefs.StartCreation[
            paginateOption: simple,
            wantHeadingHandles: ((sourceLeftHeading # NIL)
            OR (sourceRightHeading # NIL)),
            wantFootingHandles: ((sourceLeftFooting # NIL)
            OR (sourceRightFooting # NIL)),
            initialFontProps: @fontProps, initialParaProps: @paraProps,
            initialPageProps: @pageProps];
          IF targetLeftHeading # NIL THEN
            CopyHeadFootings[
              sourceLeftHeading, sourceRightHeading, sourceLeftFooting,
              sourceRightFooting, targetLeftHeading, targetRightHeading,
              targetLeftFooting, targetRightFooting];
          FreeHeadFootText[@sourceLeftHeading, z];
          FreeHeadFootText[@sourceRightHeading, z];
          FreeHeadFootText[@sourceLeftFooting, z];
          FreeHeadFootText[@sourceRightFooting, z];
          IF paraProps.tabStops.BASE # NIL THEN z.FREE[@paraProps.tabStops.BASE];
          IF pageProps.columnWidths # NIL THEN z.FREE[@pageProps.columnWidths];
          diCtxt ← [sourceDoc, targetDoc, TRUE, TRUE, FALSE, FALSE];
          [] ← DocInterchangeDefs.Enumerate[
            [doc[h: sourceDoc]], diEnumProcs, @diCtxt];
          IF diCtxt.error THEN {finalStatus ← failure; GOTO Aborted};
          IF diCtxt.aborted THEN {finalStatus ← aborted; GOTO Aborted};
          CopyFillinOrderAndModeProps[sourceDoc, targetDoc];
          DocInterchangeDefs.Close[@sourceDoc];
          [docFile, targetDocSession, ] ← DocInterchangeDefs.FinishCreation[
            @targetDoc, CheckAbort];
          IF NOT diCtxt.aborted THEN
            WrapUpFiling[docFileRef, docFile, targetDocSession];
          IF background THEN NSFile.Logoff[targetDocSession];
          EXITS
            Aborted = > {
              DocInterchangeDefs.Close[@sourceDoc];
              DocInterchangeDefs.AbortCreation[@targetDoc];
              };
          };
          }
        ELSE {PostOpenError[openStatus]; finalStatus ← failure; };
        IF background THEN {
          NSFile.Logoff[sourceDocSession]; [] ← BusyIcon.MakeUnbusy[docFileRef]; };
        }; -- CallBack

    -- start of MakeDoc
    IF background THEN {
      Process.SetPriority[Process.priorityBackground];
      [] ← BackgroundProcess.ManageMe[name: @nameRB, callBackProc: CallBack];
      }
    ELSE [] ← CallBack[NIL];
    }; -- MakeDoc

  FreeHeadFootText: PROC [
    hf: LONG POINTER TO HeadFootText, zone: UNCOUNTED ZONE] = {
    IF hf↑ # NIL THEN {
      FOR i: CARDINAL IN [0..hf.length) DO
        WITH hfBound: hf.list[i] SELECT FROM
```

```
            np =>
              IF hfBound.paraProps.tabStops.BASE # NIL THEN
                zone.FREE[@hfBound.paraProps.tabStops.BASE];
            text => XString.FreeReaderBytes[r: @hfBound.rb, z: zone];
            ENDCASE;
          ENDLOOP;
        zone.FREE[hf];
        };
      }; -- FreeHeadFootText

    CopyFillinOrderAndModeProps: PROC [
      sourceDoc, targetDoc: DocInterchangeDefs.Doc] = {
      modeProps: DocInterchangePropsDefs.ModePropsRecord;
      DocInterchangeDefs.ClearFillInOrder[targetDoc];
      DocInterchangeDefs.EnumerateFillInOrder[
        sourceDoc, AddToFillInOrder, targetDoc];
      DocInterchangeDefs.GetModeProps[sourceDoc, @modeProps];
      DocInterchangeDefs.SetModeProps[
        targetDoc, @modeProps, [
        structureShowing: TRUE, nonPrintingShowing: TRUE,
        coverSheetShowing: TRUE, promptFields: TRUE]];
      }; -- CopyFillinOrderAndModeProps

    CopyHeadFootings: PROC [
      sourceLeftHeading, sourceRightHeading, sourceLeftFooting, sourceRightFooting:
        HeadFootText,
      targetLeftHeading, targetRightHeading: DocInterchangeDefs.Heading,
      targetLeftFooting, targetRightFooting: DocInterchangeDefs.Footing] = {
      targetTC: DocInterchangeDefs.TextContainer ← [spare1[0]];

      HitNP: DocInterchangeDefs.NewParagraphProc = {
        DocInterchangeDefs.AppendNewParagraph[
          to: targetTC, paraProps: paraProps, fontProps: fontProps];
        }; -- HitNP

      HitText: DocInterchangeDefs.TextProc = {
        DocInterchangeDefs.AppendText[
          to: targetTC, text: text, textEndContext: textEndContext,
          fontProps: fontProps];
        }; -- HitText

      EnumerateHeadFoot: PROC [hf: HeadFootText] = {
        IF hf # NIL THEN {
          FOR i: CARDINAL IN [0..hf.length) DO
            WITH hfBound: hf[i] SELECT FROM
              np =>
                [] ← HitNP[
                  clientData: NIL, fontProps: @hfBound.fontProps,
                  paraProps: @hfBound.paraProps];
              text =>
                [] ← HitText[
                  clientData: NIL, fontProps: @hfBound.fontProps,
                  text: @hfBound.rb,
                  textEndContext: hfBound.textEndContext];
              ENDCASE;
            ENDLOOP;
            };
         };

      -- start of CopyHeadFootings
      targetTC ← [heading[targetLeftHeading]];
      EnumerateHeadFoot[sourceLeftHeading];
      DocInterchangeDefs.ReleaseHeading[@targetLeftHeading];
      targetTC ← [heading[targetRightHeading]];
      EnumerateHeadFoot[sourceRightHeading];
      DocInterchangeDefs.ReleaseHeading[@targetRightHeading];
      targetTC ← [footing[targetLeftFooting]];
      EnumerateHeadFoot[sourceLeftFooting];
      DocInterchangeDefs.ReleaseFooting[@targetLeftFooting];
      targetTC ← [footing[targetRightFooting]];
      EnumerateHeadFoot[sourceRightFooting];
      DocInterchangeDefs.ReleaseFooting[@targetRightFooting];
      }; -- CopyHeadFootings

   PostOpenError: PROC [status: DocInterchangeDefs.OpenStatus] = {
      rb: XString.ReaderBody ←
         SELECT status FROM
            notLocal = > XString.FromSTRING["notLocal"L, TRUE],
            outOfDiskSpace = > XString.FromSTRING["outOfDiskSpace"L, TRUE],
            outOfVM = > XString.FromSTRING["outOfVM"L, TRUE],
            busy = > XString.FromSTRING["busy"L, TRUE],
            ENDCASE = > XString.nullReaderBody;
      Attention.Post[s: @rb];
      }; -- PostOpenError

   -- Copy contents of current selection to new doc.
   MakeDocMenuCmdProc: MenuData.MenuProc = {
      IF Selection.CanYouConvert[file] THEN {
         selValue: Selection.Value ← Selection.Convert[file];
         fileRef: NSFile.Reference = LOOPHOLE[selValue.value, LONG POINTER TO
            NSFile.Reference] ↑ ;
         IF LOOPHOLE[itemData, LONG CARDINAL] = 0 THEN {
            Selection.Clear[];
            [] ← BusyIcon.MakeBusy[fileRef];
            Process.Detach[FORK MakeDoc[fileRef, TRUE]];
            }
         ELSE MakeDoc[fileRef, FALSE];
         }
      ELSE UserTerminal.BlinkDisplay[];
      }; -- MakeDocMenuCmdProc

   AddToFillInOrder: DocInterchangeDefs.FillInOrderProc = {
      targetDoc: DocInterchangeDefs.Doc = clientData;
      DocInterchangeDefs.AppendItemToFillInOrder[
         targetDoc, fillInOrderItemName, itemType];
      }; -- AddToFillInOrder

   << Called when an anchored frame was encountered in the source document.
      Copies the frame and its contents to the target document. >>
   AppendAnchoredFrameToTargetDoc: DocInterchangeDefs.AnchoredFrameProc = {
      -- use other interfaces here
      }; -- AppendAnchoredFrameToTargetDoc
```

```
AppendBreakToTargetDoc: DocInterchangeDefs.SpareProc = {
   diCtxt: DICtxtHandle = clientData;
   bp: DocInterchangeExtra1Defs.ReadonlyBreakPropsForEnum = data;
   DocInterchangeExtra1Defs.AppendBreak[
      diCtxt.targetDoc, bp.breakProps, bp.fontProps];
   }; -- AppendBreakToTargetDoc

AppendFieldToTargetDoc: DocInterchangeDefs.FieldProc = {
   diCtxt: DICtxtHandle = clientData;
   procs: DocInterchangeDefs.EnumProcsRecord ← [
      newParagraphProc: AppendNewParToNewField,
      textProc: AppendTextToNewField];
   newField: DocInterchangeDefs.Field;

   AppendNewParToNewField: DocInterchangeDefs.NewParagraphProc = {
      DocInterchangeDefs.AppendNewParagraph[
         [field[h: newField]], paraProps, fontProps]};

   AppendTextToNewField: DocInterchangeDefs.TextProc = {
      DocInterchangeDefs.AppendText[
         [field[h: newField]], text, textEndContext, fontProps]};

   IF (diCtxt.aborted ← BackgroundProcess.UserAbort[]) THEN {
      BackgroundProcess.ResetUserAbort[]; RETURN[stop: TRUE]};
   newField ← DocInterchangeDefs.AppendField[
      [doc[h: diCtxt.targetDoc]], fieldProps, fontProps];
   [] ← DocInterchangeDefs.Enumerate[[field[h: field]], @procs];
   DocInterchangeDefs.ReleaseField[@newField];
   }; -- AppendFieldToTargetDoc

AppendNewParToTargetDoc: DocInterchangeDefs.NewParagraphProc = {
   diCtxt: DICtxtHandle = clientData;
   IF diCtxt.ignoreNewPar THEN diCtxt.ignoreNewPar ← FALSE
   ELSE
      DocInterchangeDefs.AppendNewParagraph[
         [doc[h: diCtxt.targetDoc]], paraProps, fontProps];
   }; -- AppendNewParToTargetDoc

AppendPFCToTargetDoc: DocInterchangeDefs.PFCProc = {
   diCtxt: DICtxtHandle = clientData;
   IF diCtxt.ignorePFC THEN diCtxt.ignorePFC ← FALSE
   ELSE {
      targetLeftHeading, targetRightHeading: DocInterchangeDefs.Heading;
      targetLeftFooting, targetRightFooting: DocInterchangeDefs.Footing;
      procs: DocInterchangeDefs.EnumProcsRecord ← [
         newParagraphProc: HitNP, textProc: HitText];
      targetTC: DocInterchangeDefs.TextContainer ← [spare1[0]];

      HitNP: DocInterchangeDefs.NewParagraphProc = {
         DocInterchangeDefs.AppendNewParagraph[
            to: targetTC, paraProps: paraProps, fontProps: fontProps];
         }; -- HitNP

      HitText: DocInterchangeDefs.TextProc = {
         DocInterchangeDefs.AppendText[
            to: targetTC, text: text, textEndContext: textEndContext,
            fontProps: fontProps];
          }; -- HitText

        [targetLeftHeading, targetRightHeading,
          targetLeftFooting, targetRightFooting] ←
          DocInterchangeDefs.AppendPFC[
          to: diCtxt.targetDoc, pageProps: pageProps,
          wantHeadingHandles: ((leftHeading # NIL) OR (rightHeading # NIL)),
          wantFootingHandles: ((leftFooting # NIL) OR (rightFooting # NIL)),
          fontProps: fontProps];
        targetTC ← [heading[targetLeftHeading]];
        [] ← DocInterchangeDefs.Enumerate[[heading[leftHeading]], @procs];
        DocInterchangeDefs.ReleaseHeading[@targetLeftHeading];
        targetTC ← [heading[targetRightHeading]];
        [] ← DocInterchangeDefs.Enumerate[[heading[rightHeading]], @procs];
        DocInterchangeDefs.ReleaseHeading[@targetRightHeading];
        targetTC ← [footing[targetLeftFooting]];
        [] ← DocInterchangeDefs.Enumerate[[footing[leftFooting]], @procs];
        DocInterchangeDefs.ReleaseFooting[@targetLeftFooting];
        targetTC ← [footing[targetRightFooting]];
        [] ← DocInterchangeDefs.Enumerate[[footing[rightFooting]], @procs];
        DocInterchangeDefs.ReleaseFooting[@targetRightFooting];
        };
      }; -- AppendPFCToTargetDoc

    AppendTextToTargetDoc: DocInterchangeDefs.TextProc = {
      diCtxt: DICtxtHandle = clientData;
      IF (diCtxt.aborted ← BackgroundProcess.UserAbort[]) THEN {
        BackgroundProcess.ResetUserAbort[]; RETURN[stop: TRUE]; };
      DocInterchangeDefs.AppendText[
        [doc[h: diCtxt.targetDoc]], text, textEndContext, fontProps];
      }; -- AppendTextToTargetDoc

    << Copy the initial font, para, and page properties of a doc to nodes in z. >>
    GetInitialDocProps: PROC [
      docFileRef: NSFile.Reference, sourceDoc: DocInterchangeDefs.Doc,
      session: NSFile.Session, fp: DocInterchangePropsDefs.FontProps,
      pp: DocInterchangePropsDefs.ParaProps,
      pagep: DocInterchangePropsDefs.PageProps,
      sourceLeftHeading, sourceRightHeading,
      sourceLeftFooting, sourceRightFooting: LONG POINTER TO HeadFootText,
      zone: UNCOUNTED ZONE] = {
      procs: DocInterchangeDefs.EnumProcsRecord ← [
        newParagraphProc: HitNewPar, pfcProc: HitPFC];

      HitNewPar: DocInterchangeDefs.NewParagraphProc = {
        pp.basicProps ← paraProps.basicProps;
        pp.spare1 ← 0;
        IF paraProps.tabStops.LENGTH = 0 THEN pp.tabStops ← DESCRIPTOR[NIL, 0]
        ELSE {
          Storage: TYPE = RECORD [
            SEQUENCE COMPUTED CARDINAL OF DocInterchangePropsDefs.TabStop];
          pp.tabStops ← DESCRIPTOR[
            zone.NEW[Storage [paraProps.tabStops.LENGTH]],
            paraProps.tabStops.LENGTH];
        FOR ix: CARDINAL IN [0..paraProps.tabStops.LENGTH) DO
          pp.tabStops[ix] ← paraProps.tabStops[ix]; ENDLOOP;
        };
      }; -- HitNewPar

    HitPFC: DocInterchangeDefs.PFCProc = {
      fp↑ ← fontProps↑;
      pagep↑ ← pageProps↑;
      IF pageProps.unequalColumnWidths THEN {
        length: CARDINAL = pageProps.columnWidths.length;
        pagep.columnWidths ← zone.NEW[
          DocInterchangePropsDefs .ColumnWidthsRecord[length]];
        FOR i: CARDINAL IN [0..length) DO
          pagep.columnWidths.widths[i] ← pageProps.columnWidths.widths[i];
          ENDLOOP;
        pagep.columnWidths.length ← length;
        pagep.columnWidths.spare1 ← pageProps.columnWidths.spare1;
        }
      ELSE pagep.columnWidths ← NIL;
      {
      procs: DocInterchangeDefs.EnumProcsRecord ← [
        newParagraphProc: LocalNPProc, textProc: LocalTextProc];
      targetHF: LONG POINTER TO HeadFootText ← NIL;

      GrowHF: PROC [hf: LONG POINTER TO HeadFootText] = {
        IF hf↑ = NIL THEN {hf↑ ← zone.NEW[HeadFootTextRec [5]]; hf.length ← 0}
        ELSE
          IF hf.length = hf.maxLength THEN {
            new: HeadFootText = zone.NEW[HeadFootTextRec [hf.length + 5]];
            new.length ← hf.length;
            FOR i: CARDINAL IN [0..hf.length) DO new.list[i] ← hf.list[i]; ENDLOOP;
            zone.FREE[hf];
            hf↑ ← new;
            };
        }; -- GrowHF

      LocalNPProc: DocInterchangeDefs.NewParagraphProc = {
        GrowHF[targetHF];
        targetHF.list[targetHF.length] ← [
          fontProps: fontProps↑, v: np[paraProps: paraProps↑]];
        WITH hfBound: targetHF.list[targetHF.length] SELECT FROM
          np => {
            IF paraProps.tabStops.LENGTH = 0 THEN
              hfBound.paraProps.tabStops ← DESCRIPTOR[NIL, 0]
            ELSE {
              Storage: TYPE = RECORD [
                SEQUENCE COMPUTED CARDINAL OF
                DocInterchangePropsDefs.TabStop];
              hfBound.paraProps.tabStops ← DESCRIPTOR[
                zone.NEW[Storage [paraProps.tabStops.LENGTH]],
                paraProps.tabStops.LENGTH];
              FOR ix: CARDINAL IN [0..paraProps.tabStops.LENGTH) DO
                hfBound.paraProps.tabStops[ix] ← paraProps.tabStops[ix];
                ENDLOOP;
              };
            };
            ENDCASE => ERROR;
          targetHF.length ← targetHF.length + 1;
          }; -- LocalNPProc

        LocalTextProc: DocInterchangeDefs.TextProc = {
          GrowHF[targetHF];
          targetHF.list[targetHF.length] ← [
            fontProps: fontProps ↑,
            v: text[
            rb: XString.CopyToNewReaderBody[r: text, z: zone],
            textEndContext: textEndContext]];
          targetHF.length ← targetHF.length + 1;
          }; -- LocalTextProc

        -- start of HitPFC
        targetHF ← sourceLeftHeading;
        sourceLeftHeading ↑ ← NIL;
        [] ← DocInterchangeDefs.Enumerate[[heading[leftHeading]], @procs];
        targetHF ← sourceRightHeading;
        sourceRightHeading ↑ ← NIL;
        [] ← DocInterchangeDefs.Enumerate[[heading[rightHeading]], @procs];
        targetHF ← sourceLeftFooting;
        sourceLeftFooting ↑ ← NIL;
        [] ← DocInterchangeDefs.Enumerate[[footing[leftFooting]], @procs];
        targetHF ← sourceRightFooting;
        sourceRightFooting ↑ ← NIL;
        [] ← DocInterchangeDefs.Enumerate[[footing[rightFooting]], @procs];
        };
        RETURN[stop: TRUE];
        }; -- HitPFC

      [] ← DocInterchangeDefs.Enumerate[[doc[h: sourceDoc]], @procs];
      }; -- GetInitialDocProps

    << Change the name of the new document to be the same as the old doc. >>
    SetNewDocName: PROC [
      oldDoc: NSFile.Reference, newDoc: NSFile.Handle,
      newDocSession: NSFile.Session] = {
      oldDocFile: NSFile.Handle ← NSFile.OpenByReference[
        reference: oldDoc, controls: [lock: share, access: [read: TRUE]]];
      attRec: NSFile.AttributesRecord;
      attArray: ARRAY [0..0] OF NSFile.Attribute;
      NSFile.GetAttributes[
        file: oldDocFile, selections: [interpreted: [name: TRUE]], attributes: @attRec];
      attArray[0] ← [name[value: attRec.name]];
      NSFile.ChangeAttributes[
        file: newDoc, attributes: DESCRIPTOR[attArray], session: newDocSession];
      NSFile.Close[oldDocFile];
      NSFile.FreeWords[
        DESCRIPTOR[attRec.name.bytes, NSString.WordsForString[attRec.name.length]]];
      }; -- SetNewDocName

    WrapUpFiling: PROC [
      docFileRef: NSFile.Reference, docFile: NSFile.Handle,
      targetDocSession: NSFile.Session] = {
      refDoc, refDt: NSFile.Reference;
    fileDt: NSFile.Handle;
    SetNewDocName[
       oldDoc: docFileRef, newDoc: docFile, newDocSession: targetDocSession];
    refDoc ← NSFile.GetReference[docFile, targetDocSession];
    refDt ← StarDesktop.GetCurrentDesktopFile[];
    fileDt ← NSFile.OpenByReference[refDt, , targetDocSession];
    NSFile.Move[docFile, fileDt, , targetDocSession]; -- put new doc on Desktop
    NSFile.Close[fileDt, targetDocSession];
    NSFile.Close[docFile, targetDocSession];
    StarDesktop.AddReferenceToDesktop[refDoc];
    }; -- WrapUpFiling

Init: PROC = {
    name: XString.ReaderBody ←
       XString.FromSTRING["Copy Most of Doc (Forked)"L, TRUE];
    Attention.AddMenuItem[
       MenuData.CreateItem[z, @name, MakeDocMenuCmdProc]];
    name ← XString.FromSTRING["Copy Most of Doc (Notifier)"L, TRUE];
    Attention.AddMenuItem[
       MenuData.CreateItem[z, @name, MakeDocMenuCmdProc, 1]];
    }; -- Init

-- main code
Init[];
}.
```

72.4 Index of Interface Items

73

DocInterchangePropsDefs

73.1 Overview

This interface contains procedures and data types used to describe the properties in documents; it is intended for use with all the ***InterchangeDefs** interfaces.

Most records below have spare fields for future use. When specifying values for these, it is important to use zero if you do not know of a correct value to use.

73.2 Interface Items

73.2.1 Frame Properties

The chief type in this section is the **FramePropsRecord**, which describes the properties of an anchored frame.

```
FrameProps: TYPE = LONG POINTER TO FramePropsRecord;

ReadonlyFrameProps: TYPE = LONG POINTER TO READONLY FramePropsRecord;

FramePropsRecord: TYPE = RECORD [
   borderStyle: BorderStyle,
   borderThickness: CARDINAL,
   frameDims: FrameDims,
   fixedWidth,
   fixedHeight: BOOL,
   span: Span,
   verticalAlignment: VerticalAlignment,
   horizontalAlignment: HorizontalAlignment,
   topMarginHeight,
   bottomMarginHeight,
   leftMarginWidth,
   rightMarginWidth: CARDINAL,
   spare1: LONG CARDINAL];
```

```
BorderStyle: TYPE = MACHINE DEPENDENT {
    invisible(0),solid, dashed, broken, dotted, double, firstAvailable, lastAvailable(255)};
```

borderStyle specifies the characteristics of the lines that make up the frame border.

borderThickness specifies the thickness of the frame border. This value is in units of 1/72 inch. Note that **borderThickness** depends on the **borderStyle** specified: for **double** borders **borderThickness** can range from 3 to 18 in multiples of 3 points (a "point" is 1/72 inch), while for all other **borderStyles** **borderThickness** can range from 1 to 6 points.

```
FrameDims: TYPE = RECORD [w, h: CARDINAL];
```

frameDims specifies the height and width of the frame. These dimensions are also in units of 1/72 inch.

fixedWidth and **fixedHeight** indicate whether the frame will expand when necessary.

```
Span: TYPE = MACHINE DEPENDENT {partialColumn(0),
    fullColumn, partialPage, fullPage, firstAvailable, lastAvailable(255)};
```

span specifies how much of the page the frame occupies. Currently only **fullColumn** and **fullPage** are supported.

```
VerticalAlignment: TYPE = MACHINE DEPENDENT {
    top(0), bottom, floating, firstAvailable, lastAvailable(255)};

HorizontalAlignment: TYPE = MACHINE DEPENDENT {
    left(0), centered, right, floating, firstAvailable, lastAvailable(255)};
```

vertical and **horizontalAlignment** specify the alignment of the frame relative to the page.

topMarginHeight, **bottomMarginHeight**, **leftMarginWidth**, and **rightMarginWidth** are the margins of the frame, in units of 1/72 inch.

all items marked as **spare** are for future use.

73.2.2 Page Properties

The chief type in this section is the **PagePropsRecord**, which describes the various properties that can be associated with a page in a ViewPoint document.

```
PageProps: TYPE = LONG POINTER TO PagePropsRecord;

ReadonlyPageProps: TYPE = LONG POINTER TO READONLY PagePropsRecord;

PagePropsRecord: TYPE = RECORD [
    pageDims: PageDims, --layout
    topMarginHeight,
    bottomMarginHeight,
    leftMarginWidth,
    rightMarginWidth: CARDINAL,
    startingPageSide: PageSide,
    bindingMarginWidth: CARDINAL,
    nColumns: CARDINAL, -- column structure
    balancedColumns, unequalColumnWidths: BOOL,
    columnSpacing: CARDINAL,
    columnWidths: ColumnWidths,
    startingPageNumber: CARDINAL, --page numbering
    pageNumberFormat: NumberFormat,
    restartPageNumbering: BOOL,
    startingLineNumber, -- line numbering
    lineNumberInterval: CARDINAL,
    lineNumberFormat: NumberFormat,
    lineNumberLocation: LineNumberLocation,
    headingStartsOnThisPage, -- heading
    headingSameOnLeftRightPages,
    footingStartsOnThisPage, -- footing
    footingSameOnLeftRightPages: BOOL,
    spare1: LONG CARDINAL];

PageDims: TYPE = MACHINE DEPENDENT RECORD [w, h: CARDINAL];

PageSide: TYPE = MACHINE DEPENDENT {
    nil(0), left, right, firstAvailable, lastAvailable(255)};
```

pageDims are the width and height of the table, in units of 1/72 inch. **topMarginHeight**, **bottomMarginHeight**, **leftMarginWidth**, and **rightMarginWidth** describe the page margins; these values are also in units of 1/72 inch. **startingPageSide** indicates whether the first page of the document should be a left-hand page or a right-hand page; **nil** means that there is no difference between the two. **bindingMarginWidth** is the width of the binding margin, if there is one.

nColumns, **balancedColumns**, **unequalColumnWidth**, and **columnSpacing** determine column structure. **nColumns** is the number of columns; **balancedColumns** specifies whether the length of the column is equal to the length of the page. **unequalColumnWidth** indicates that the columns may have varying widths. **columnSpacing** is the amount of space between columns, in units of 1/72 inch. A maximum of 50 column widths may be specified.

```
ColumnWidths: TYPE = LONG POINTER TO ColumnWidthsRecord;

ColumnWidthsRecord: TYPE = RECORD [
    length: CARDINAL,
    spare1: LONG CARDINAL,
    widths: SEQUENCE maxLength: CARDINAL OF ColumnWidthRecord];

ColumnWidthRecord: TYPE = RECORD [
    w: CARDINAL,
    spare1: LONG CARDINAL];
```

The **ColumnWidthsRecord** record contains the number of columns (**length**) and a sequence that contains the width of each column. Spare fields are included in both the record and each of the sequence elements.

startingPageNumbers, pageNumberFormat, and restartPageNumbering describe the page numbering properties. startingPageNumber indicates the page number at which the numbering should start; restartPageNumbering specifies whether renumbering should restart for this page, or continue from where the last numbering left off.

```
NumberFormat: TYPE = MACHINE DEPENDENT {
    cardinal(0), lowerCaseLetter, upperCaseLetter, lowerCaseRoman,
    upperCaseRoman, firstAvailable, lastAvailable(255)};
```

pageNumberFormat specifies the format of the page number; currently only cardinal is implemented.

```
LineNumberLocation: TYPE = MACHINE DEPENDENT {
    leftMargin(0), rightMargin, outerMargin, bothMargins,
    firstAvailable,lastAvailable(255)};
```

startingLineNumber, lineNumberInterval, lineNumberFormat, and lineNumberLocation are not currently implemented.

The remaining properties describe headings and footings. headingStartsOnThisPage and footingStartsOnThisPage indicate whether the designated heading/footing should start on this page or the next; headingSameOnLeftRightPage and footingSameOnLeftRightPages specifies whether all pages have the same heading/footing. spare1 is for future use.

73.2.3 Field Properties

The chief field property is the FieldPropsRecord, which describes the properties of a field.

```
FieldProps: TYPE = LONG POINTER TO FieldPropsRecord;
```

```
ReadonlyFieldProps: TYPE = LONG POINTER TO READONLY FieldPropsRecord;
```

```
FieldPropsRecord: TYPE = RECORD [
    language: MultiNational.Language,
    length: CARDINAL,
    required: BOOL,
    skipIf: SkipIfChoiceType,
    stopOnSkip: BOOL,
    type: FieldChoiceType,
    fillInRule,
    description,
    format,
    name,
    range,
    skipIfField: XString.ReaderBody,
    fillInRuleRuns: FontRuns,
    spare1: LONG CARDINAL];
```

language determines the format of date and amount fields. There are many formats, so you would have to check the format for each particular language.

length specifies the maximum number of logical characters the field may contain.

required indicates whether the user is required to fill in the field. If **required** is TRUE, the user will not be able to use NEXT or SKIP to advance to the next field until this field has a value.

```
SkipIfChoiceType: TYPE = MACHINE DEPENDENT {
    empty(0), notEmpty, never, always, firstAvailable, lastAvailable(255)};
```

skipIf defines the conditions under which the field can be skipped when the user presses the NEXT key. **stopOnSkip** specifies whether the skipping action should stop at this field or not.

```
FieldChoiceType: TYPE = MACHINE DEPENDENT {
    any(0), text, amount, date, firstAvailable, lastAvailable(255)};
```

type specifies the type of data that can be in the field. **any** indicates that the field can contain any characters, including frames (but not other fields). **text** indicates that the field can contain only letters, digits, and symbols entered from the keyboard. **amount** indicates that the field can contain only numbers, spaces, and the following symbols: +__ * $, . (). **date** specifies that entries in the field can contain only a date.

fillInRule defines the fill-in rule for this field.

description is posted for each field entered with the NEXT key if the document is set to prompt for fields. **format** controls the format in which information is presented. For a **type** of **text**, this property defines a required pattern that must be matched. For a **type** of **amount** or **date**, this field controls the form in which the contents of the field are presented, regardless of how the user enters them. For a **type** of **any**, the format property is not used.

name is the name of the field. If no name is provided, the field will automatically be named Field*n*, as in Field1, Field2, and so on.

range defines a specific range of acceptable entries.

skipIfField contains the name of the field that will appear in the property sheet Skip if field.

fillInRuleRuns is an auxiliary data structure that the client can attach to the **XString.Reader** that describes the fill-in rule for the field. A font run describes the subsequences of characters within a **Reader** that have the same font attributes.

73.2.4 Font Properties

The **FontPropsRecord** is the chief type in this Section. Section 73.2.4.1 describes the **fontDesc** field; section 73.2.4.2 describes the other fields in a **FontPropsRecord**.

```
FontProps: TYPE = LONG POINTER TO FontPropsRecord;

ReadonlyFontProps: TYPE = LONG POINTER TO READONLY FontPropsRecord;

FontPropsRecord: TYPE = RECORD [
    fontDesc: FontDescription,
    offset: INTEGER,
    foregroundBackground: ForegroundBackground,
    nUnderlines: CARDINAL,
    strikeout: BOOL,
    placement: Placement,
    toBeDeleted,
    revised: BOOL,
    width: Width,
    spare1: LONG CARDINAL];
```

73.2.4.1 FontDescription

```
FontDescription: TYPE = RECORD [
    family: Family,
    designVariant: DesignVariant,
    posture: Posture,
    weight: Weight,
    pointSize: CARDINAL,
    serifness: Serifness,
    spare1: LONG CARDINAL];

Family: TYPE = MACHINE DEPENDENT {
    century(0), frutiger(1), titan(2), pica(3), trojan(4), vintage(5), elite(6),
    letterGothic(7), master(8), cubic(9), roman(10), scientific(11), gothic(12),
    bold(13), ocrB(14), spokesman(15), xeroxLogo(16), centuryThin(17),
    scientificThin(18), helvetica(19), helveticaCondensed(20), optima(21),
    times(22), baskerville(23), spartan(24), bodoni(25), palatino(26),
    caledonia(27), memphis(28), excelsior(29), olympian(30), univers(31),
    universCondensed(32), trend(33), boxPS(34), terminal(35), ocrA(36), logo1(37),
    logo2(38), logo3(39), geneva2(40), times2(41), square3(42), courier(43),
    futura(44), prestige(45), aLLetterGothic(46), centurySchoolBook(47),
    firstUnused(48), lastUnused(510), backstop(511)};
```

family is the font family.

```
DesignVariant: TYPE = MACHINE DEPENDENT {
    null(0), roman, italic, firstAvailable, lastAvailable(255)};
```

designVariant specifies whether the character is roman or italic. **null** is not currently a valid value.

```
Posture: TYPE = MACHINE DEPENDENT {
    null(0), upright, slanted, backslanted, firstAvailable, lastAvailable(255)};
```

posture indicates the slant (stress) of the character, if any. **null** is not currently a valid value.

```
Weight: TYPE = MACHINE DEPENDENT {
    null(0), ultraLight, extraLight, light, semiLight, medium, semiBold, bold,
    extraBold, ultraBold, firstAvailable, lastAvailable(255)};
```

weight is the thickness of the character.

pointSize is the size of the font. Note that this value must be in the subrange [0..1023].

```
Serifness: TYPE = MACHINE DEPENDENT {
    null(0), serif, sansSerif, firstAvailable, lastAvailable(255)};
```

serifness indicates whether or not the character has serifs. **null** is not currently a valid value.

spare1 is for future use.

73.2.4.2 The other fields in FontPropsRecord

offset is the offset of the character from the baseline.

```
ForegroundBackground: TYPE = MACHINE DEPENDENT {
    null(0), blackOnWhite, whiteOnBlack, firstAvailable, lastAvailable(255)};
```

foregroundBackground indicates the color of the character relative to the display.

nUnderlines indicates the number of times that the character is underlined; the value must be in the range [0..2].

strikeout indicates whether or not the character has been struck through.

```
Placement: TYPE = MACHINE DEPENDENT{
    null(0), sub, subSub, subSuper, super, superSub, superSuper, userSpecified,
    firstAvailable, lastAvailable(255)};
```

placement indicates the position of the character relative to the line.

toBeDeleted indicates "normal" text that has been marked for deletion in Redlining mode.

revised indicates text that was typed in while Redlining was on but before finalizing the Redlined revisions.

```
Width: TYPE = MACHINE DEPENDENT {
    proportional(0), quarter, third, half, threeQuarters, full, firstAvailable,
    lastAvailable(255)};
```

width is used for Japanese text and should be set to **proportional** to get normal characters.

73.2.5 Font Runs

FontRuns are used to associate font properties with text. **XString** provides no facilities for associating font properties with text; **DocInterchangePropsDefs** allows the client to create font information structures that point into **XString** structures to make the association.

The data structures in this section mark font runs, which are consecutive characters with the same font. A **FontRunsRec** describes the font, while a cardinal value describes where the font starts in the text.

In addition, this interface allows the client to enumerate the font runs in a given **XString** body of text.

```
Run: TYPE = RECORD [
    props: FontPropsRecord,
    index: CARDINAL,
    context: XString.Context,
    spare1: LONG CARDINAL];
```

A **Run** indicates the beginning of a font run. **props** is the field describing the font used in the font run. **index** is the byte offset in the byte sequence that holds the text; it is the byte offset from the beginning of the byte sequence to the byte after the byte run. **context** is the **XString** context describing the next byte run. The context of the first byte run is contained in the reader body. See the next section for further explanation.

```
FontRuns: TYPE = LONG POINTER TO FontRunsRec;
```

```
FontRunsRec: TYPE = RECORD [
    length: CARDINAL,
    spare1: LONG CARDINAL,
    runs: SEQUENCE maxLength: CARDINAL OF Run];
```

FontRuns points to **FontRunsRec**, which is a record containing a sequence of **Runs**.

```
EnumerateFontRuns: PROC [
    r: XString.Reader,
    runs: FontRuns,
    proc: FontRunProc,
    clientData: LONG POINTER ← NIL]
    RETURNS [stopped: BOOL];
```

```
FontRunProc: TYPE = PROC [
    r: XString.Reader,
    fontProps: FontProps,
    clientData: LONG POINTER]
    RETURNS [stop: BOOL ← FALSE];
```

EnumerateFontRuns allows you to perform some action for each font run in an **XString.Reader**. **FontRunProc** is a call-back procedure that you pass to **EnumerateFontRuns**. If **FontRunProc** returns **stopped = TRUE**, the enumeration stops and **EnumerateFontRuns** returns **stopped = TRUE**. **clientData** is client defined data that you pass to **EnumerateFontRuns**, which passes it to **FontRunProc** every time **FontRunProc** is invoked.

73.2.5.1 Meaning of Index and Context Fields in Run

As stated earlier, **index** is the index into the XString of the byte following that run. **context** is the **XString.Context** in effect after that run. Here are two examples:

A ReaderBody with offset = 0, limit = 12, with bytes *abcdefghijkl*; font runs that describe the first three bytes as *fontA*, the next four as *fontB*, and the last five as *fontC* would be:

```
fontRun 0: [props: fontA, index: 3, context: ...]
fontRun 1: [props: fontB, index: 7, context: ...]
fontRun 2: [props: fontC, index: 12, context: ...]
```

A ReaderBody with offset = 7, limit = 19, with bytes *abcdefghijkl*; font runs that describe the first three bytes as *fontA*, the next four as *fontB*, and the last five as *fontC* would be:

```
fontRun 0: [props: fontA, index: 10, context: ...]
fontRun 1: [props: fontB, index: 14, context: ...]
fontRun 2: [props: fontC, index: 19, context: ...]
```

73.2.6 Paragraph Properties

The chief type in this section is the **ParaPropsRecord**, which describes the possible paragraph properties.

```
ParaProps: TYPE = LONG POINTER TO ParaPropsRecord;

ReadonlyParaProps: TYPE = LONG POINTER TO READONLY ParaPropsRecord;

ParaPropsRecord: TYPE = RECORD [
    basicProps: BasicPropsRecord,
    tabStops: TabStops,
    spare1: LONG CARDINAL];
```

basicProps describes all standard paragraph properties (those on the Paragraph property sheet); **tabStops** describes the current tab settings (the information on the Tab Settings property sheet).

The following sections describe the **BasicPropsRecord** and **TabStops** records in detail.

73.2.6.1 BasicPropsRecord

```
BasicProps: TYPE = LONG POINTER TO BasicPropsRecord;

ReadonlyBasicProps: TYPE = LONG POINTER TO READONLY BasicPropsRecord;

BasicPropsRecord: TYPE = RECORD [
    preLeading,
    postLeading,
    leftIndent,
    rightIndent,
    lineHeight: CARDINAL,
    paraAlignment: ParaAlignment,
    justified,
    hyphenated,
    keepWithNextPara: BOOL,
    language: MultiNational.Language,
    streakSuccession: StreakSuccession,
    defaultTabStopSpacing: DefaultTabStopSpacing,
    defaultTabStopAlignment: TabStopAlignment,
    spare1: LONG CARDINAL];
```

preLeading and **postLeading** are the spacing before and after the paragraph respectively; these values are in units of 1/72 inch.

leftIndent and **rightIndent** are the left and right paragraph margins; these values are in units of 1/72 inch.

lineHeight is the default line height for the paragraph; this value is in units of 1/72 inch.

```
ParaAlignment: TYPE = MACHINE DEPENDENT {
    left(0), center, right, firstAvailable, lastAvailable(255)};
```

paraAlignment indicates the alignment of the paragraph relative to the containing text column or text block.

justified when TRUE, causes the text in the paragraph to stretch to make a straight right edge.

hyphenated indicates whether the paragraph will be hypenated at the end of lines to improve justification.

keepWithNextPara indicates whether the pagination operation should attempt to keep this paragraph on the same page or column as the next one.

language is the language for the paragraph; this information is used for formatting decimal tabs. It is also used when items are added to the paragraph (e.g., a field inherits the paragraph language when added to the paragraph).

```
StreakSuccession: TYPE = MACHINE DEPENDENT {
    leftToRight(0), rightToLeft, firstAvailable, lastAvailable(255)};
```

streakSuccession specifies whether a "streak" of characters should logically be read from left to right (e.g. English) or right to left (e.g. Hebrew).

```
TabStopOffset: TYPE = CARDINAL;

DefaultTabStopSpacing: TYPE = CARDINAL;
```

defaultTabStopSpacing is the default number of spaces between tabs.

```
TabStopAlignment: TYPE = MACHINE DEPENDENT {
    left(0), center, right, decimal, firstAvailable, lastAvailable(255)};
```

defaultTabStopAlignment is the default alignment for tabs: tabs can be relative to the left paragraph margin, the center of the paragraph, the right paragraph margin, or decimal points.

spare1 is for future use.

73.2.6.2 Tabs

```
TabStops: TYPE = LONG DESCRIPTOR FOR ARRAY OF TabStop;

TabStop: TYPE = RECORD [
    dotLeader: BOOLEAN,
    tabStopOffset: TabStopOffset,
    tabStopAlignment: TabStopAlignment,
    spare1: LONG CARDINAL];
```

tabStops describes the currently set tabs for the paragraph.

dotLeader indicates whether the tab has leader dots. **tabStopOffset** indicates the location of the tab, relative to the paragraph margin. **tabStopAlignment** indicates the alignment of the tab.

Any array of tabstops used to create or modify a document object must be sorted in increasing order of **tabStopOffsets**. A **tabStopOffset** that is equal to a previous one is ignored. During enumeration, tabstop arrays passed to the client will always be sorted in this manner. The maximum number of tabstops that may be set in any paragraph is **nTabsMax**.

```
nTabsMax: CARDINAL = 100;
```

nTabsMax is the maximum number of tabs that there can be in a paragraph.

73.2.7 Mode Properties

Mode properties describe the commands in the document and auxillary menus of a ViewPoint document.

```
ModeProps: TYPE = LONG POINTER TO ModePropsRecord;

ReadonlyModeProps: TYPE = LONG POINTER TO READONLY ModePropsRecord;

ModePropsRecord: TYPE = RECORD [
    structureShowing,
    nonPrintingShowing,
    coverSheetShowing,
    promptFields: BOOL,
    spare1: LONG CARDINAL];
```

structureShowing, nonPrintingShowing, coverSheetShowing, and promptFields specify the appearance of the displayed document.

```
BooleanFalseDefault: TYPE = BOOL ← FALSE;

ModeSelections: TYPE = PACKED ARRAY ModeElements OF BooleanFalseDefault;
```

```
ModeElements: TYPE = {
    structureShowing, nonPrintingShowing, coverSheetShowing, promptFields,
    spare1, spare2, spare3, spare4, spare5, spare6, spare7, spare8};
```

ModeSelections are used to specify which **ModeElements** of a document should be acted upon.

73.2.8 Constants

The **null*Props** constants are declared so that clients may initialize property records with "neutral" properties. In most cases, each field value is the same as what would be set by the corresponding **Get*PropsDefaults** operation (sec 73.2.8).

```
nullFrameProps: FramePropsRecord = [
    borderStyle: solid,
    borderThickness: 2,
    frameDims: [72, 72],
    fixedWidth: FALSE,
    fixedHeight: FALSE,
    span: partialColumn,
    verticalAlignment: floating,
    horizontalAlignment: centered,
    topMarginHeight: 0,
    bottomMarginHeight: 0,
    leftMarginWidth: 0,
    rightMarginWidth: 0,
    spare1: 0];

nullPageProps: PagePropsRecord = [
    pageDims: [0, 0],
    topMarginHeight: 0,
    bottomMarginHeight: 0,
    leftMarginWidth: 0,
    rightMarginWidth: 0,
    startingPageSide: left,
    bindingMarginWidth: 0,
    nColumns: 1,
    balancedColumns: FALSE,
    unequalColumnWidths: FALSE,
    columnSpacing: 0,
    columnWidths: NIL,
    startingPageNumber: 1,
    pageNumberFormat: cardinal,
    restartPageNumbering: FALSE,
    startingLineNumber: 1,
    lineNumberInterval: 1,
    lineNumberFormat: cardinal,
    lineNumberLocation: leftMargin,
    headingStartsOnThisPage: TRUE,
    headingSameOnLeftRightPages: TRUE,
    footingStartsOnThisPage: TRUE,
    footingSameOnLeftRightPages: TRUE,
    spare1: 0];

nullColumnWidth: ColumnWidthRecord = [
    w: 0,
    spare1: 0];

nullFieldProps: FieldPropsRecord = [
    language: USEnglish,
    length: 0,
    required: FALSE,
    skipIf: never,
    stopOnSkip: FALSE,
    type: any,
    fillInRule: XString.nullReaderBody,
    description: XString.nullReaderBody,
    format: XString.nullReaderBody,
    name: XString.nullReaderBody,
    range: XString.nullReaderBody,
    skipIfField: XString.nullReaderBody,
    fillInRuleRuns: NIL,
    spare1: 0];

nullFontProps: FontPropsRecord = [
    fontDesc: nullFontDescription,
    offset: 0,
    foregroundBackground: blackOnWhite,
    nUnderlines: 0,
    strikeout: FALSE,
    placement: null,
    toBeDeleted: FALSE,
    revised: FALSE,
    width: proportional,
    spare1: 0];

nullFontDescription: FontDescription = [
    family: modern,
    designVariant: roman,
    posture: upright,
    weight: medium,
    pointSize: 12,
    serifness: sansSerif,
    spare1: 0];

nullRun: Run = [
    props: nullFontProps,
    index: 0,
    context: XString.unknownContext,
    spare1: 0];

classic: Family = century;
```

```
modern: Family = frutiger;

nullParaProps: ParaPropsRecord = [
    basicProps: nullBasicProps,
    tabStops: DESCRIPTOR[NIL, 0],
    spare1: 0];

nullBasicProps: BasicPropsRecord = [
    preLeading: 0,
    postLeading: 0,
    leftIndent: 0,
    rightIndent: 0,
    lineHeight: 12,
    paraAlignment: left,
    justified: FALSE,
    hyphenated: FALSE,
    keepWithNextPara: FALSE,
    language: USEnglish,
    streakSuccession: leftToRight,
    defaultTabStopSpacing: 18,
    defaultTabStopAlignment: left,
    spare1: 0];

nullTabStop: TabStop = [
    dotLeader: FALSE,
    tabStopOffset: 0,
    tabStopAlignment: left,
    spare1: 0];

nullModeProps: ModePropsRecord = [
    structureShowing: FALSE,
    nonPrintingShowing: FALSE,
    coverSheetShowing: FALSE,
    promptFields: FALSE,
    spare1: 0];
```

73.2.9 Default Properties

The **Get*PropsDefaults** procedures are called to obtain default values for property fields. These procedures differ from the constants in that they may obtain information from the user profile. Before calling any of these procedures, the client must declare a record of the appropriate type and pass its address to the **Get*PropsDefaults** procedure. None of these procedures allocate any additional data that the client would later have to free.

```
GetFramePropsDefaults: PROC [props: FrameProps];

GetPagePropsDefaults: PROC [props: PageProps];

GetFieldPropsDefaults: PROC [props: FieldProps];

GetFontPropsDefaults: PROC [props: FontProps];
```

```
GetParaPropsDefaults: PROC [props: ParaProps];

GetModePropsDefaults: PROC [props: ModeProps];

GetPageNumberDelimiter: PROC RETURNS [XChar.Character];
```

73.3 Index of Interface Items

Item	Page
BasicProps: TYPE	9
BasicPropsRecord: TYPE	9
BooleanFalseDefault: TYPE	11
BorderStyle: TYPE	2
classic: Family	13
ColumnWidthRecord: TYPE	3
ColumnWidths: TYPE	3
ColumnWidthsRecord: TYPE	3
DefaultTabStopSpacing: TYPE	10
DesignVariant: TYPE	6
EnumerateFontRuns: PROC	8
Family: TYPE	6
FieldChoiceType: TYPE	5
FieldProps: TYPE	4
FieldPropsRecord: TYPE	4
FontDescription: TYPE	6
FontProps: TYPE	5
FontPropsRecord: TYPE	5
FontRunProc: TYPE	8
FontRuns: TYPE	8
FontRunsRec: TYPE	8
ForegroundBackground: TYPE	7
FrameDims: TYPE	2
FrameProps: TYPE	1
FramePropsRecord: TYPE	1
GetFieldPropsDefaults: PROC	14
GetFontPropsDefaults: PROC	14
GetFramePropsDefaults: PROC	14
GetModePropsDefaults: PROC	15
GetPageNumberDelimiter: PROC	15
GetPagePropsDefaults: PROC	14
GetParaPropsDefaults: PROC	15
HorizontalAlignment: TYPE	2
LineNumberLocation: TYPE	4
ModeElements: TYPE	12
ModeProps: TYPE	11
ModePropsRecord: TYPE	11
modern: Family	14
ModeSelections: TYPE	11
nTabsMax: CARDINAL	11
nullBasicProps: BasicPropsRecord	14
nullColumnWidth: ColumnWidthRecord	13
nullFieldProps: FieldPropsRecord	13
nullFontDescription: FontDescription	13
nullFontProps: FontPropsRecord	13
nullFrameProps: FramePropsRecord	12
nullModeProps: ModePropsRecord	14
nullPageProps: PagePropsRecord	12
nullParaProps: ParaPropsRecord	14
nullRun: Run	13
nullTabStop: TabStop	14
NumberFormat: TYPE	4
PageDims: TYPE	3
PageProps: TYPE	2
PagePropsRecord: TYPE	2
PageSide: TYPE	3
ParaAlignment: TYPE	10
ParaProps: TYPE	9
ParaPropsRecord: TYPE	9
Placement: TYPE	7
Posture: TYPE	6
ReadonlyBasicProps: TYPE	9
ReadonlyFieldProps: TYPE	4
ReadonlyFontProps: TYPE	5
ReadonlyFrameProps: TYPE	1
ReadonlyModeProps: TYPE	11
ReadonlyPageProps: TYPE	2
ReadonlyParaProps: TYPE	[illegible]
Run: TYPE	[illegible]
Serifness: TYPE	7
SkipIfChoiceType: TYPE	5
Span: TYPE	2
StreakSuccession: TYPE	10
TabStop: TYPE	11
TabStopAlignment: TYPE	10
TabStopOffset: TYPE	10
TabStops: TYPE	11
VerticalAlignment: TYPE	2
Weight: TYPE	6
Width: TYPE	7

74

EquationInterchangeDefs

74.1 Overview

EquationInterchangeDefs provides utilities for creating and enumerating the content of anchored equation frames. It is meant to be used in conjunction with **DocInterchangeDefs**.

An equation is a container for sub-parts, each of which is either a single object (like a character or parenthesis) or an object that, like an equation, contains sub-parts itself. Examples of the latter include fractions, integrals, and matrices. The data structure that represents this "thing that contains other things" is the **Handle**, defined below. A handle is created by calling **StartEquation** and is passed to most of the routines in this interface. A typical scenario for creating a document with an equation frame in it would be:

```
doc ← DocInterchangeDefs.StartCreation[...];
h ← StartEquation[doc];
Append...[h, ...]; -- add an object to h. Ref. any Append* routine below.
[equation, ...] ← FinishEquation[h];
[...] ← DocInterchangeDefs.AppendAnchoredFrame[
   to: doc, type: equation, content: equation];
[...] ← DocInterchangeDefs.FinishCreation[@doc];
```

All the **Append*** routines defined below add an object to the end of a container's list of objects.

Most records below have spare fields for future use. When specifying values for these, it is important to use zero if you do not know of a correct value to use.

74.2 Interface Items

74.2.1 General data types

```
Handle: TYPE = LONG POINTER TO Object;

Object: TYPE = RECORD [
   zone: UNCOUNTED ZONE,
   equation: DocInterchangeDefs.Instance,
    spare1: LONG CARDINAL,
    private: ARRAY [0..0) OF WORD];
```

The **zone** is a normal type of scratch zone that will be destroyed when DocInterchangeDefs.FinishCreation or DocInterchangeDefs.AbortCreation has been called on the document. Clients are free to use this zone for node allocation.

equation uniquely identifies an equation or an equation sub-part within a document.

```
Looks: TYPE = MACHINE DEPENDENT {normal(0), bold, italic, lastAvailable(255)};

Size: TYPE = MACHINE DEPENDENT {
    smallest(0), small(1), regular(2), large(3), lastAvailable(255)};

EquationCharProps: TYPE = LONG POINTER TO EquationCharPropsRecord;

ReadonlyEquationCharProps: TYPE = LONG POINTER TO READONLY
    EquationCharPropsRecord;

EquationCharPropsRecord: TYPE = RECORD [
    looks: Looks,
    size: Size,
    spares: ARRAY [0..8) OF WORD];
```

74.2.2 Equation creation

```
StartEquation: PROC [doc: DocInterchangeDefs.Doc] RETURNS [Handle];
```

Creates a handle that, when passed to **FinishEquation**, will yield the content of an anchored equation frame. The content of the frame is to be passed to DocInterchangeDefs.AppendAnchoredFrame. See FinishEquation. May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM].

```
EquationProc: TYPE = PROC [h: Handle, clientData: LONG POINTER];
```

Procedures of this type are passed to certain **Append*** routines (the routines that create objects that contain other equation objects). The **EquationInterchange** implementation then calls this procedure back in order to fill in a particular sub-part. The client proc should do this by calling various **Append*** routines with the passed handle. See the **Append*** routines.

```
Side: TYPE = {left, right};
```

Specifies a side for objects like parentheses, brackets, and braces.

```
AppendBrace: PROC [h: Handle, side: Side, props: ReadonlyEquationCharProps];
```

Appends a curly brace to the handle. **props.looks** should always be **normal**. May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM, badParameter] (badParameter if props.looks # normal).

```
AppendBracket: PROC [h: Handle, side: Side, props: ReadonlyEquationCharProps];
```

Appends a square bracket to the handle. **props.looks** should always be **normal**. May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM, badParameter] (badParameter if props.looks # normal).

```
AppendCharacter: PROC [
    h: Handle, character: XChar.Character, props: ReadonlyEquationCharProps];
```

Appends a character to the handle. May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM].

```
AppendString: PROC [
    h: Handle, string: XString.Reader, props: ReadonlyEquationCharProps];
```

Appends a string to the handle. Each character of the string is given the designated props. May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM].

```
AppendFraction: PROC [
    h: Handle, size: Size, numerator, denominator: EquationProc,
    clientData: LONG POINTER];
```

Appends a fraction to the handle. During the call to **AppendFraction**, **numerator** and **denominator** are called back. Each should fill in their respective parts of the fraction via more **Append*** routines (passing the handle that was passed to *them*). May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM].

```
IntegralType: TYPE = MACHINE DEPENDENT{
    integral(0), lineIntegral(1), verticalBar(2), lastAvailable(255)};
```

```
AppendIntegral: PROC [
    h: Handle, props: ReadonlyEquationCharProps, type: IntegralType,
    lowerBound, upperBound: EquationProc, clientData: LONG POINTER];
```

Appends an integral to the handle. **props.looks** should always be **normal**. During the call to **AppendIntegral**, **lowerBound** and **upperBound** are called back. Each should fill in their respective parts of the integral via more **Append*** routines (passing the handle that was passed to *them*). May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM, badParameter] (badParameter if props.looks # normal).

```
AppendLimit: PROC [
    h: Handle, props: ReadonlyEquationCharProps, range: EquationProc,
    clientData: LONG POINTER];
```

Appends a limit to the handle. **props.looks** should always be **normal**. During the call to **AppendLimit**, **range** is called back. It should fill in the range of the integral via more **Append*** routines (passing the handle that was passed to **range**). May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM, badParameter] (badParameter if props.looks # normal).

```
Clearance: TYPE = [-63..64]; -- units are 1/72 inch.
```

```
ColumnAlignment: TYPE = MACHINE DEPENDENT {
    left(0), right(1), centered(2), decimal(3), lastAvailable(255)};

RowAlignment: TYPE = MACHINE DEPENDENT {
    top(0), bottom(1), centered(2), onbaseline(3), lastAvailable(255)};

NextOrder: TYPE = MACHINE DEPENDENT {byrow(0), bycol(1), lastAvailable(255)};

EquationMatrixProps: TYPE = LONG POINTER TO EquationMatrixPropsRecord;

ReadonlyEquationMatrixProps: TYPE = LONG POINTER TO READONLY
    EquationMatrixPropsRecord;

EquationMatrixPropsRecord: TYPE = RECORD [
    size: Size,
    ctCols, ctRows: CARDINAL [1..256],
    clearanceInterCol, clearanceInterRow: Clearance,
    rowalignment: RowAlignment,
    colalignment: ColumnAlignment,
    nextorder: NextOrder,
    spare1: LONG CARDINAL];

MatrixCellProc: TYPE = PROC [
    h: Handle, row, column: CARDINAL [1..256], clientData: LONG POINTER];

AppendMatrix: PROC [
    h: Handle, props: ReadonlyEquationMatrixProps, proc: MatrixCellProc,
    clientData: LONG POINTER];
```

Appends a matrix to the handle. During the call to **AppendMatrix**, **proc** is called back once for each cell. The **proc** should fill in the appropriate cell of the matrix via more **Append*** routines (passing the handle that was passed to **proc**). The client's **proc** should not assume it is being called in any particular order relative to other cells in the same matrix. May raise **DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM]**.

```
AppendOther: PROC [h: Handle, data: LONG UNSPECIFIED, clientData: LONG POINTER];
```

AppendOther is for future use. It is currently unimplemented.

```
AppendOverBar: PROC [
    h: Handle, size: Size, equation: EquationProc, clientData: LONG POINTER];
```

Appends an overbar to the handle (this is just a horizontal bar over an equation sub-part). During the call to **AppendOverBar**, **equation** is called back. It should fill in the pieces underneath the overbar via more **Append*** routines (passing the handle that was passed to **equation**). May raise **DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM]**.

```
AppendParenthesis: PROC [h: Handle, side: Side, props: ReadonlyEquationCharProps];
```

Appends a parenthesis to the handle. props.looks should always be normal. May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM, badParameter] (badParameter if props.looks # normal).

```
AppendScript: PROC [
    h: Handle, size: Size, base, superScript, subScript: EquationProc,
    clientData: LONG POINTER];
```

Appends a base/superscript/subscript thingy to the handle. During the call to AppendScript, base, superScript and subScript are called back. Each should fill in their respective parts of the whatsit via more Append* routines (passing the handle that was passed to *them*). May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM].

```
SummationType: TYPE = MACHINE DEPENDENT {
    sum(0), product(1), union(2), intersection(3), lastAvailable(255)};

AppendSummation: PROC [
    h: Handle, props: ReadonlyEquationCharProps, type: SummationType,
    lowerBound, upperBound: EquationProc, clientData: LONG POINTER];
```

Appends a summation structure to the handle. props.looks should always be normal. During the call to AppendSummation, lowerBound and upperBound are called back. Each should fill in their respective parts of the summation sign via more Append* routines (passing the handle that was passed to *them*). May raise DocInterchangeDefs.Error[documentFull, readonlyDoc, outOfDiskSpace, outOfVM, badParameter] (badParameter if props.looks # normal).

```
FinishEquation: PROC [h: Handle]
    RETURNS [
        equation: DocInterchangeDefs.Instance,
        equationHeight, equationWidth: LONG CARDINAL];
```

Finishes an equation and returns an instance for its content. This should be passed to the content parameter of DocInterchangeDefs.AppendAnchoredFrame. equationHeight and equationWidth are for informative purposes only (they are in 1/72 inch units).

74.2.3 Equation enumeration

```
BraceProc: TYPE = PROC [
    side: Side, props: ReadonlyEquationCharProps, clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

BracketProc: TYPE = PROC [
    side: Side, props: ReadonlyEquationCharProps, clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

CharacterProc: TYPE = PROC [
    character: XChar.Character, props: ReadonlyEquationCharProps,
    clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];
```

```
FractionProc: TYPE = PROC [
    size: Size, numerator, denominator: DocInterchangeDefs.Instance,
    clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

IntegralProc: TYPE = PROC [
    props: ReadonlyEquationCharProps, type: IntegralType,
    lowerBound, upperBound: DocInterchangeDefs.Instance, clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

LimitProc: TYPE = PROC [
    props: ReadonlyEquationCharProps, element: DocInterchangeDefs.Instance,
    clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

MatrixElementProc: TYPE = PROC [
    element: DocInterchangeDefs.Instance, row, column: CARDINAL [1..256],
    clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

EnumerateMatrixProc: TYPE = PROC [proc: MatrixElementProc, clientData: LONG POINTER]
    RETURNS [stopped: BOOLEAN ← FALSE];

MatrixProc: TYPE = PROC [
    props: ReadonlyEquationMatrixProps, matrixProc: EnumerateMatrixProc,
    clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

OtherObjectType: TYPE = MACHINE DEPENDENT {firstAvailable(0), lastAvailable(255)};

OtherProc: TYPE = PROC [
    clientData: LONG POINTER, instance: DocInterchangeDefs.Instance,
    objectType: OtherObjectType]
    RETURNS [stop: BOOLEAN ← FALSE];
```

The OtherProc is currently unimplemented--it will not be called during an enumeration.

```
OverBarProc: TYPE = PROC [
    size: Size, element: DocInterchangeDefs.Instance, clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

ParenthesisProc: TYPE = PROC [
    side: Side, props: ReadonlyEquationCharProps, clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

ScriptProc: TYPE = PROC [
    size: Size, base, superScript, subScript: DocInterchangeDefs.Instance,
    clientData: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

SummationProc: TYPE = PROC [
    props: ReadonlyEquationCharProps, type: SummationType,
     lowerBound, upperBound: DocInterchangeDefs.Instance, clientData: LONG POINTER]
     RETURNS [stop: BOOLEAN ← FALSE];

EnumProcs: TYPE = LONG POINTER TO EnumProcsRecord;

EnumProcsRecord: TYPE = RECORD [
     brace: BraceProc ← NIL,
     bracket: BracketProc ← NIL,
     character: CharacterProc ← NIL,
     fraction: FractionProc ← NIL,
     integral: IntegralProc ← NIL,
     limit: LimitProc ← NIL,
     matrix: MatrixProc ← NIL,
     other: OtherProc ← NIL,
     overBar: OverBarProc ← NIL,
     parenthesis: ParenthesisProc ← NIL,
     script: ScriptProc ← NIL,
     summation: SummationProc ← NIL];

EnumerateEquation: PROC [
     equation: DocInterchangeDefs.Instance, procs: EnumProcs, clientData: LONG POINTER]
     RETURNS [stopped: BOOLEAN ← FALSE];
```

Enumerates an equation. Note that the **Instance** could be that of an anchored equation frame or that of some sub-part. If it's the former, then it should be the **content** parameter passed to a **DocInterchangeDefs.AnchoredFrameProc**. Do NOT pass **anchoredFrame** (another **Instance** parameter of a **DocInterchangeDefs.AnchoredFrameProc**) to **EnumerateEquation**.

74.3 Index of Interface Items

Item	Page
AppendBrace: PROC	2
AppendBracket: PROC	2
AppendCharacter: PROC	3
AppendFraction: PROC	3
AppendIntegral: PROC	3
AppendLimit: PROC	3
AppendMatrix: PROC	4
AppendOther: PROC	4
AppendOverBar: PROC	4
AppendParenthesis: PROC	4
AppendScript: PROC	5
AppendString: PROC	3
AppendSummation: PROC	5
BraceProc: TYPE	5
BracketProc: TYPE	5
CharacterProc: TYPE	5
Clearance: TYPE	3
ColumnAlignment: TYPE	4
EnumerateEquation: PROC	7
EnumerateMatrixProc: TYPE	6
EnumProcs: TYPE	7
EnumProcsRecord: TYPE	7
EquationCharProps: TYPE	2
EquationCharPropsRecord: TYPE	2
EquationMatrixProps: TYPE	4
EquationMatrixPropsRecord: TYPE	4
EquationProc: TYPE	2
FinishEquation: PROC	5
FractionProc: TYPE	6
Handle: TYPE	1
IntegralProc: TYPE	6
IntegralType: TYPE	3
LimitProc: TYPE	6
Looks: TYPE	2
MatrixCellProc: TYPE	4
MatrixElementProc: TYPE	6
MatrixProc: TYPE	6
NextOrder: TYPE	4
Object: TYPE	1
OtherObjectType: TYPE	6
OtherProc: TYPE	6
OverBarProc: TYPE	6
ParenthesisProc: TYPE	6
ReadonlyEquationCharProps: TYPE	2
ReadonlyEquationMatrixProps: TYPE	4
RowAlignment: TYPE	4
ScriptProc: TYPE	6
Side: TYPE	2
Size: TYPE	2
StartEquation: PROC	2
SummationProc: TYPE	6
SummationType: TYPE	5

GraphicsInterchangeDefs

75.1 Overview

GraphicsInterchangeDefs provides utilities for creating and enumerating the contents of anchored and nested graphics frames. It is intended to be used in conjunction with **DocInterchangeDefs**.

75.1.1 Creating Graphics

To create new graphics, the client starts by calling **StartGraphics**, which initializes a graphics frame so that information can be added to it. This procedure returns a **Handle**, which is a pointer to an opaque type that contains, among other things, a graphics container. A graphics container is just an object that can contain graphic objects: a graphics container can be an anchored graphics frame, a nested graphics frame, a cusp button within a graphics frame, or another similar construct, such as a chart.

Once the client has a **Handle**, it can pass that **Handle** to various **Add*** routines to add new graphics objects, such as curves, rectangles, bitmaps, and text frames, to the graphics frame.

The client can also add nested frames, such as non-anchored graphics frames, cusp buttons, or graphics clusters, to the anchored frame. To create these structures, the client should call **StartGraphicsFrame, StartCuspButton**, or **StartCluster**, respectively. Each of these procedures takes a graphics container as a parameter, and returns another graphics handle. The client can then use this as the graphics container in other calls to **Add*** routines.

When everything has been added to a graphics container, the final step is to call a **Finish*** routine: **FinishGraphics**, **FinishButton**, **FinishGraphicsFrame**, or **FinishCluster**. **FinishGraphics** returns a graphics handle that can be passed to **DocInterchangeDefs**.

Thus, the scenario for creating a document with a floating graphics frame nested within an anchored graphics frame looks something like this:

1. Call **DocInterchangeDefs.StartCreation** to get a document handle (**doc**).

2. Call **StartGraphics[doc]** to get an anchored frame handle (**h**).
3. Call **Add*[h]** to add graphics to the anchored frame.
4. Call **StartGraphicsFrame** to get a handle for a nested graphics frame (**gfh**).
5. Call **Add*[gfh]** to add graphics to the nested frame.
6. Call **FinishGrapicsFrame[gfh]** to finish the nested frame.
7. Call **FinishGraphics[h]** to complete the anchored frame and get an object of type **DocInterchangeDefs.Instance (graphics)**.
8. Call **DocInterchangeDefs.AppendAnchoredFrame[graphics]**.
9. Call **DocInterchangeDefs.FinishCreation[@doc]**.

75.1.2 Reading Graphics

GraphicsInterchangeDefs also includes the facilities to read the contents of graphics frames. To read a graphics frame, the client should call **Enumerate**. **Enumerate** takes as parameters a graphics container and a record of call back procedures, one for each of the following graphics objects: {bitmap frame, cusp button, cluster, curve, ellipse, form field, frame, image, line, point, rectangle, text, triangle, other}.

Enumerate reads the contents of the graphics container, calling the appropriate procedure for each object that it encounters. If the client does not provide a procedure for a particular type of object, objects of that type will be ignored. Each of the client-supplied enumeration procedures can stop the enumeration if it so desires.

There are similar procedures to enumerate the contents of cusp buttons. **EnumerateButtonProgram** takes a button program and a record to handle the various objects that can be in a button program: new paragraphs and text.

75.2 Interface Items

75.2.1 Creating graphics

After calling a **Start*** routine to initialize a graphics container, the client will typically call various **Add*** routines to add information to the graphics container. The **Add*** routines defined below add an object to the end of the list of objects in the specified graphics container.

Many property records defined below have spare fields for future use. When specifying values for these, it is important to use zero if you do not know of a correct value to use.

The operations for creating graphics are divided into eight sections, which are:

1. Start routines, which describes the operations for creating graphic containers
2. Setting extra frame properties, for dealing with additional properties that only anchored frames have

3. Adding geometrics, for adding simple graphics objects such as curves, ellipses, lines, points, rectangles, and triangles
4. Adding frames, for adding all types of nested frames
5. Adding to a cusp button, for specifying a cusp button's program
6. Adding miscellaneous objects, for adding objects not defined in this interface
7. Release routines, which describes operations similar to the Release* operations in DocInterchangeDefs
8. Finish routines, for wrapping up graphic containers

75.2.1.1 Start routines

To create new graphics objects, the client must first call **StartGraphics** to get an anchored frame handle.

```
StartGraphics: PROC [
    doc: DocInterchangeDefs.Doc]
    RETURNS [h: Handle];
```

StartGraphics creates a new graphics frame within **doc**.

```
Handle: TYPE = LONG POINTER TO Object;
Object: TYPE;
```

There are also similar routines to create nested frames within a graphics container. **StartCluster**, **StartGraphicsFrame**, and **StartButton** each initialize a nested frame within a graphics container. All **Start** routines return a **Handle**, which the client can then pass to the various **Add*** routines to add graphics to that graphics container.

```
StartCluster: PROC [h: Handle, box: Box] RETURNS [ch: Handle];
```

StartCluster initializes a set of graphics objects in **h**. **box** describes the size and location of the cluster relative to the anchored frame; **place** and **dims** are in micas.

```
Box: TYPE = RECORD [place: Place, dims: Dims]; -- micas

Place: TYPE = RECORD [x, y: LONG INTEGER];

Dims: TYPE = RECORD [w, h: LONG INTEGER];
```

Note that an object's place is always relative to the object that contains it. A place of [0, 0] indicates the container's upper left corner. When the container is a graphics frame, the upper left corner is the one that includes the margins (even when the frame is anchored). Increasing X values indicate more rightward locations, and increasing Y values indicate more *downward* locations. Although an object's dims are declared as LONG INTEGERS, it is illegal to specify negative width or height for an object's box. An object's place should therefore always indicate its upper left corner.

Every graphic object, including points and containers, has a **box**. An object's **box.dims** define the size of the object. If a client specifies a **box** that is too small for an object in a call to an **Add*** routine, then only that part of the object which lies inside the **box** will be displayed.

When graphic objects are encountered during enumeration, the data passed to the client procedure always follows the above rules.

```
StartGraphicsFrame: PROC [
    h: Handle,
    box: Box,
    frameProps: ReadonlyFrameProps,
    name, description: XString.Reader ← NIL,
    spareProps: LONG POINTER ← NIL,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        gfh: Handle,
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

StartGraphicsFrame initializes a nested graphics frame in **h**. **box** indicates the size and location of the nested frame relative to the graphics container; these values are in micas.

frameProps are the properties for the frame.

```
FrameProps: TYPE = LONG POINTER TO FramePropsRec;

ReadonlyFrameProps: TYPE = LONG POINTER TO READONLY FramePropsRec;

FramePropsRec: TYPE = RECORD [
    brush: Brush,
    fixedShape: BOOL,
    margins: ARRAY Side OF LONG CARDINAL,
    captionContent: ARRAY Side OF DocInterchangeDefs.Caption,
    spare1: LONG CARDINAL];

Brush: TYPE = RECORD [
    wthbrush: LONG CARDINAL,
    stylebrush: StyleBrush];

StyleBrush: TYPE = MACHINE DEPENDENT{
    invisible(0), solid(1), dashed(2), dotted(3), double(4), broken(5), (15)};
```

brush describes the properties of the lines that make up the frame. The brush width is in micas. The standard brush widths on the property sheet are 35, 71, 106, 141, 176 and 212. Note that **wthbrush** depends on the **stylebrush** specified: for **double** borders **wthbrush** should be 3 times the usual width.

fixedShape indicates whether the frame will expand in a uniform fashion.

margins are the frame margins, in points.

```
Side: TYPE = {top, bottom, left, right};
```

captionContent is an array of captions associated with the frame. Note that the **captionContent** parameter is only meaningful during enumeration, and not during **Start** or **Add*** routines, since the caption content is added after the frame is created.

spare1 is for future use.

name and **description** are the name and description of the graphics frame as it appears in the property sheet.

spareProps is for future use.

want*CaptionHandle indicates whether the client wants the frame to have the corresponding captions. If the client passes TRUE for one of these values, the corresponding return value will be non-NIL. The client can then use **DocInterchangeDefs** routines to add text to the caption. Note that the caption must eventually be freed with **DocInterchangeDefs.ReleaseCaption**.

gfh is a handle to the newly created graphics frame.

```
StartButton: PROC [
   h: Handle,
   box: Box,
   buttonProps: ReadonlyButtonProps,
   frameProps: ReadonlyFrameProps,
   wantProgramHandle,
   wantTopCaptionHandle,
   wantBottomCaptionHandle,
   wantLeftCaptionHandle,
   wantRightCaptionHandle: BOOLEAN ← FALSE]
   RETURNS [
      bfh: Handle,
      buttonProgram: ButtonProgram,
      topCaption, bottomCaption,
      leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

StartButton initializes a cusp button as a graphics container. **box** describes the size and location of the cusp button relative to the graphics container **h**.

```
ButtonProps: TYPE = LONG POINTER TO ButtonPropsRec;

ReadonlyButtonProps: TYPE = LONG POINTER TO READONLY ButtonPropsRec;

ButtonPropsRec: TYPE = RECORD [
   name: XString.Reader,
   spare1: LONG CARDINAL];

ButtonProgram: TYPE = LONG POINTER TO ButtonProgramObject;

ButtonProgramObject: TYPE;
```

buttonProps are the default properties for the button. If the client defaults this parameter, **StartButton** will generate a new unique name for the button.

wantProgramHandle specifies whether the client wants to be able to add to the button's program. If the client specifies TRUE, and is returned a valid button program handle, then it must later free that handle with a call to **ReleaseButtonProgram** (see section 75.2.1.7). **GraphicsInterchangeDefs** provides several procedures that the client can use to add data to the cusp program; see section 75.2.1.5, *Adding to a cusp button*, for information on these procedures.

All other properties are as described for **StartGraphicsFrame**.

75.2.1.2 Setting extra frame properties

```
SetExtraAnchoredFrameProps: PROC [
    doc: DocInterchangeDefs.Doc,
    anchoredFrame: DocInterchangeDefs.Instance,
    name, description: XString.Reader,
    spareProps: LONG POINTER ← NIL,
    selections: ExtraAnchoredFramePropsSelections];

ExtraAnchoredFramePropsSelections: TYPE = PACKED ARRAY
    ExtraAnchoredFramePropsElements OF DocInterchangePropsDefs.BooleanFalseDefault;

ExtraAnchoredFramePropsElements: TYPE = {name, description, spareProps};
```

The client can associate a name and description with an anchored frame by calling **SetExtraAnchoredFrameProps**. **doc** is the document that contains the anchored frame. **anchoredFrame** is the frame in which the client intends to add a name or description. **spareProps** are for future use and should be left defaulted to NIL. **selections** indicate which properties the client intends to add.

75.2.1.3 Adding geometrics to a graphics container

```
AddCurve: PROC [h: Handle, box: Box, curveProps: ReadonlyCurveProps];

CurveProps: TYPE = LONG POINTER TO CurvePropsRec;

ReadonlyCurveProps: TYPE = LONG POINTER TO READONLY CurvePropsRec;
CurvePropsRec: TYPE = RECORD [
    brush: Brush,
    lineEndNW: LineEnd,
    lineEndSE: LineEnd,
    lineEndHeadNW: LineEndHead,
    lineEndHeadSE: LineEndHead,
    direction: LineDirection,
    placeNW, placeApex, placeSE, placePeak: Place,
    fixedAngle: BOOL,
    spare1: LONG CARDINAL];

LineEnd: TYPE = MACHINE DEPENDENT {
    flush(0), square(1), round(2), arrow(3), (7)};
```

```
LineEndHead: TYPE = MACHINE DEPENDENT {
    none(0), h1(1), h2(2), h3(3), (15)};

LineDirection: TYPE = MACHINE DEPENDENT {
    WE(0), NS(1), NwSe(2), SwNe(3)};
```

AddCurve adds the curve described by **curveProps** to the specified graphics container. **box** specifies the location of the curve relative to the graphics frame. If **box.dims** is smaller than the curve, only that part of the curve that fits within **box.dims** will be displayed.

brush indicates the line properties of the curve; **brush** is as described earlier for **StartGraphicsFrame.**

lineEnd* describe the properties of the ends of the curve. **lineEndNW** describes the end that would paint first if the curve is traced clockwise; **lineEndSE** describes the end that would paint last tracing clockwise (Figure 75.1).

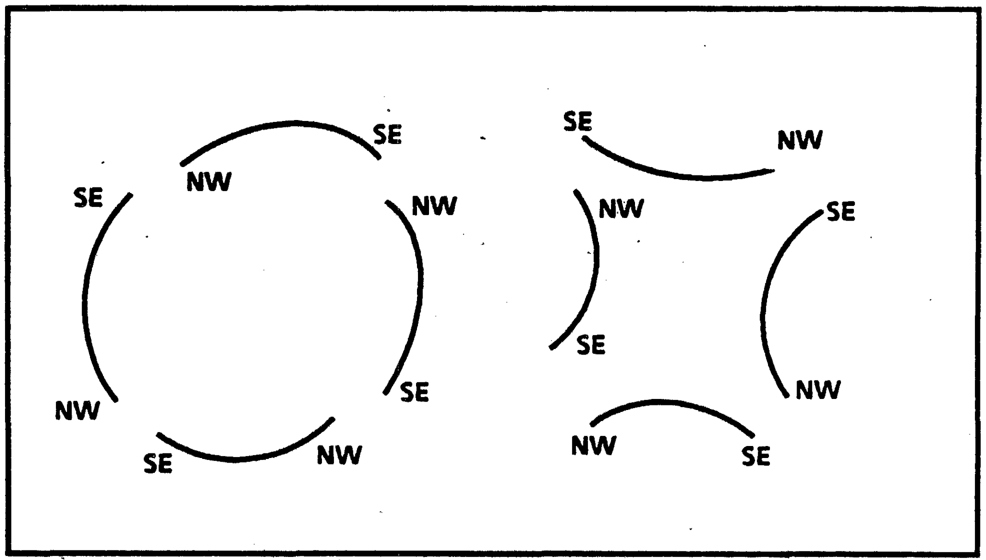


Figure 75.1 Curve Direction

If **lineEnd** = **arrow**, then **lineEndHead** describes the type of arrow: **h1** is the thinnest arrowhead; **h3** is the thickest as shown in Figure 75.2. If **lineEnd** ≠ **arrow**, then **lineEndHead** should be **none.**

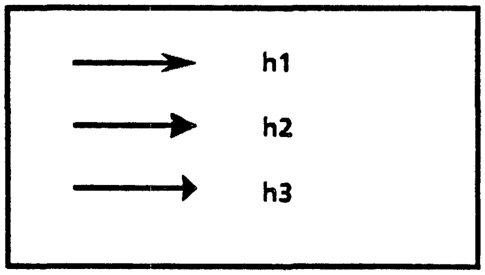


Figure 75.2 Arrowheads

direction is ignored; the client should always set this to **WE**.

place* defines the curve by specifying its endpoints, apex, and peak. These points are relative to **box**, and not the frame itself. Recall that curves paint clockwise; clients must ensure that the **NW** endpoint appears before the **SE** endpoint when tracing the curve clockwise. Figure 75.3 illustrates these four points for two different curves; the triangle marks the apex, the square marks the peak, and the circles mark the endpoints.

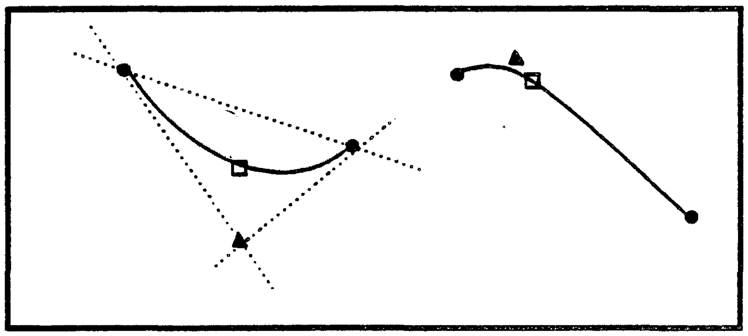


Figure 75.3 Defining curves

fixedAngle indicates that the curve will maintain its shape when grown or shrunk. **spare1** is for future use.

```
AddEccentricCurve: PROC [
    h: Handle,
    box: Box,
    eccentricCurveProps: ReadonlyEccentricCurveProps];

EccentricCurveProps: TYPE = LONG POINTER TO EccentricCurvePropsRec;

ReadonlyEccentricCurveProps: TYPE =
    LONG POINTER TO READONLY EccentricCurvePropsRec;

EccentricCurvePropsRec: TYPE = RECORD [
    brush: Brush,
    lineEndNW: LineEnd,
    lineEndSE: LineEnd,
    lineEndHeadNW: LineEndHead,
    lineEndHeadSE: LineEndHead,
    direction: LineDirection,
    placeNW, placeApex, placeSE: Place,
    eccentricity: CARDINAL,
    fixedAngle: BOOL,
    spare1: LONG CARDINAL];
```

AddEccentricCurve is just like **AddCurve** except that the curve is specified by its endpoints, apex, and eccentricity, rather than by endpoints, apex and peak.

eccentricity is a fraction represented by **eccentricity / LAST[CARDINAL]** . This allows the highest possible precision for eccentricities between 0 and 1.

```
AddEllipse: PROC [
    h: Handle,
    box: Box,
    ellipseProps: ReadonlyEllipseProps];

EllipseProps: TYPE = LONG POINTER TO EllipsePropsRec;

ReadonlyEllipseProps: TYPE = LONG POINTER TO READONLY EllipsePropsRec;

EllipsePropsRec: TYPE = RECORD [
    brush: Brush,
    shading: Shading,
    fixedShape: BOOL,
    spare1: LONG CARDINAL];
```

AddEllipse adds an ellipse to the specified graphics container. **box.dims** determine the size and shape of the ellipse; **box.place** determines its location relative to the container **h**.

```
Shading: TYPE = RECORD [gray: Gray, textures: Textures];

Gray: TYPE = MACHINE DEPENDENT {
    none(0), gray25(1), gray50(2), gray75(3), black(4), (15)};

Textures: TYPE = PACKED ARRAY Texture OF BOOLEAN;

Texture: TYPE = MACHINE DEPENDENT {
    vertical(0), horizontal(1), nwse(2), swne(3), polkadot(4), (11)};
```

Within the **EllipseProps**, **brush** describes the ellipses' border, and **shading** describes its interior. The shading of the interior can be 25%, 50%, or 75% gray or solid black; the texture can be horizontal, vertical, or diagonal lines, or dots. **fixedShape** has the same meaning for all shapes: it indicates that the proportions of the object will not change when the user grows or shrinks it. **spare1** is for future use for all shapes.

```
AddLine: PROC [
    h: Handle,
    box: Box,
    lineProps: ReadonlyLineProps];

LineProps: TYPE = LONG POINTER TO LinePropsRec;

ReadonlyLineProps: TYPE = LONG POINTER TO READONLY LinePropsRec;

LinePropsRec: TYPE = RECORD [
    brush: Brush,
    lineEndNW: LineEnd,
    lineEndSE: LineEnd,
    lineEndHeadNW: LineEndHead,
    lineEndHeadSE: LineEndHead,
    direction: LineDirection,
    fixedAngle: BOOL,
    spare1: LONG CARDINAL];
```

AddLine adds a line to the graphics container **h**. **box** describes the bounding box of the line. **LineProps** are all as described above for curves.

```
AddPieslice: PROC [
    h: Handle,
    box: Box,
    piesliceProps: ReadonlyPiesliceProps];

PiesliceProps: TYPE = LONG POINTER TO PieslicePropsRec;

ReadonlyPiesliceProps: TYPE = LONG POINTER TO READONLY PieslicePropsRec;

PieslicePropsRec: TYPE = RECORD [
    brush: Brush,
    shading: Shading,
    center, start, stop: Place,
    spare1: LONG CARDINAL];
```

AddPieslice adds a pieslice object to the graphics container **h**. **box** describes the bounding box of the pieslice. **PiesliceProps** are all as described above for ellipses. **center**, **start**, and **stop** are all relative to **box.place**. The arc of a pieslice goes from **start** to **stop** in a clockwise direction. Only brush styles of **none** and **solid** are supported. All the pieslice properties and operations are currently defined in **GraphicsInterchangeExtra3Defs**.

```
AddPoint: PROC [
    h: Handle,
    box: Box,
    pointProps: ReadonlyPointProps];

PointProps: TYPE = LONG POINTER TO PointPropsRec;

ReadonlyPointProps: TYPE = LONG POINTER TO READONLY PointPropsRec;

PointPropsRec: TYPE = RECORD [
    wthbrush: LONG CARDINAL,
    pointStyle: PointStyle,
    pointFill: PointFill,
    spare1: LONG CARDINAL];

PointStyle: TYPE = MACHINE DEPENDENT {round(0), square(1), triangle(2), cross(3), (255)};

PointFill: TYPE = MACHINE DEPENDENT {solid(0), hollow(1), (255)};
```

AddPoint adds the point described by **box** and **pointProps** to the graphics container **h**. **wthbrush** is in micas. **pointStyle** and **pointFill** are as shown in the Point object property sheet.

```
AddRectangle: PROC [
    h: Handle,
    box: Box,
    rectangleProps: ReadonlyRectangleProps];
```

```
RectangleProps: TYPE = LONG POINTER TO RectanglePropsRec;

ReadonlyRectangleProps: TYPE = LONG POINTER TO READONLY RectanglePropsRec;

RectanglePropsRec: TYPE = RECORD [
    brush: Brush,
    shading: Shading,
    fixedShape: BOOL,
    spare1: LONG CARDINAL];
```

AddRectangle adds the rectangle specified by **box** to the graphics container **h**. Rectangle properties are as described above for ellipses.

```
AddTriangle: PROC [
    h: Handle,
    box: Box,
    triangleProps: ReadonlyTriangleProps];

TriangleProps: TYPE = LONG POINTER TO TrianglePropsRec;

ReadonlyTriangleProps: TYPE =
    LONG POINTER TO READONLY TrianglePropsRec;

TrianglePropsRec: TYPE = RECORD [
    brush: Brush,
    shading: Shading,
    place1, place2, place3: Place, -- corners of triangle
    fixedShape: BOOL,
    spare1: LONG CARDINAL];
```

AddTriangle adds a triangle to the graphics conainter **h**. **brush** and **shading** are as described for ellipses; **place1**, **place2**, and **place3** are the corners of the triangle, relative to **box**. The triangle is added to **h** at **box.place**.

75.2.1.4 Adding frames to a graphics container

The following **Add*** routines add various types of frame objects to the graphics container. Each of these routines has a parameter of type **FrameProps** that describes the frame, and **want*CaptionHandle** parameters that determine the captions for that frame. These parameters are as described in section 75.2.1.1, **StartGraphicsFrame**.

```
AddBitmap: PROC [
    h: Handle,
    box: Box,
    bitmapProps: ReadonlyBitmapProps,
    frameProps: ReadonlyFrameProps,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

bitmapProps describes a bitmap frame. **bitmapProps** largely correspond to the properties that the user sees in the property sheet.

```
BitmapProps: TYPE = LONG POINTER TO BitmapPropsRec;

ReadonlyBitmapProps: TYPE = LONG POINTER TO READONLY BitmapPropsRec;

BitmapPropsRec: TYPE = RECORD [
    opaque: BOOLEAN,
    xOffset, yOffset: LONG INTEGER,
    printFile: XString.ReaderBody,
    displaySource: BmDisplay,
    scalingProps: BitmapScalingProps,
    spare1: LONG CARDINAL];
```

opaque specifies whether the bitamp is opaque or transparent. **xOffset** and **yOffset** control the position of the bitmap within the bitmap frame. Setting both to 0 will position the bitmap flush in the upper left-hand corner. These values are in units of 1/72 inch.

printFile is the source for the bitmap to print. This is usually the same as the display source, but the client may specify a file name as an alternate print source if desired.

```
BmDisplay: TYPE = RECORD [
    SELECT type: * FROM
        internal = > [bm: LONG POINTER TO BitmapData],
        file = > [name: XString.ReaderBody],
    ENDCASE];
```

The source for the displayed bitmap is in one of two locations: either internal (the bits are copied into the document), or in a file on the desktop.

```
BitmapData: TYPE = RECORD [
    signature: INTEGER ← bmSignature, -- do not use any other value
    xScale: Interpress.Rational,
    yScale: Interpress.Rational,
    xDim: CARDINAL, -- # of bits wide
    yDim: CARDINAL, -- # of bits tall
    bpl: CARDINAL, -- Bits Per Line = ((xDim + 15) / 16) * 16
    pages: NSSegment.PageCount,
    bits: PACKED ARRAY [0..0) OF Environment.Byte];

bmSignature: INTEGER = 23456;
```

The actual bitmap is described by a **BitmapData** record. **signature** is a validity check for the bitmap. If a bitmap signature is anything but **bmSignature**, the implementation will not recognize it as a valid bitmap.

xScale and **yScale** specify the bitmap scale. At present, the only scale that is supported is 72 spots per inch, so the client should always set **xScale** and **yScale** to [254, 720,000]. (The default unit for an Interpress.Rational is meters; converting inches to meters yields 720,000 spots per 254 meters, since 1 inch = 2.54 cm.)

xDim and **yDim** describe the size of the bitmap. **bpl** is the width of the bitmap, rounded to the nearest word boundary.

pages is the number of pages that the bitmap occupies, and **bits** is the actual bitmap.

```
BitmapScalingProps: TYPE = RECORD [
    SELECT type: * FROM
    printerResolution = > [resolution: CARDINAL],
    fixed = > [
        horizontalAlignment: {center, right, left},
        verticalAlignment: {center, bottom, top},
        scalingPercentage: CARDINAL [0..1024)],
    automatic = > [shape: {similar, fillUp}],
    other = > [spare1: PACKED ARRAY [2..15) OF [0..1], spare2: CARDINAL],
    ENDCASE];
```

scalingProps specifies one of the three bitmap scaling modes: **automatic**, **fixed** or **printerResolution**. The client will generally default the mode to **automatic** with **shape = similar**; this ensures that the bitmap will be automatically magnified/shrunk to fit just inside the bitmap frame until either the vertical or horizontal edge reaches the frame's edge. If **fillUp** is specified, the vertical and horizontal scaling factors are individually determined so that the bitmap completely fills the frame.

The **fixed** mode requires the client to control the bitmap's alignment (by Alignment parameters) and scaling (by Scale parameter). The **scalingPercentage** allows the client to shrink or magnify the bitmap. A **scalingPercentage** value of 100 means that the bitmap should be displayed and printed the same size as the original. A value of 50 means that the bitmap is shrunk to one half both vertically and horizontally. **scalingPercentage** must be in the range [1..1000].

printerResolution indicates the resolution of the printer; typical values are: 72, 75, 150, 200, and 300. Other values can be specified and must be the number of spots per inch.

```
AddFormField: PROC [
    h: Handle,
    box: Box,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    frameProps: ReadonlyFrameProps,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    expandRight, expandBottom: BOOL ← FALSE,
    wantFieldHandle,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        field: DocInterchangeDefs.Field,
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

AddFormField adds the specified field to h at the location box.place. A more flexible version of this operation is now supported--see AddFormFieldX below.

If the client specifies wantFieldHandle = TRUE, AddFormField will return a DocInterchangeDefs.Field; the client must eventually free this field with a call to DocInterchangeDefs.ReleaseField. To add information to the field, the client should use the facilities of DocInterchangeDefs.

```
AddFormFieldX: PROC [
    h: Handle,
    box: Box,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    frameProps: ReadonlyFrameProps,
    textFrameProps: ReadonlyTextFrameProps,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    wantFieldHandle,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        field: DocInterchangeDefs.Field,
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

This operation is currently in GraphicsInterchangeExtra1Defs (see older form above).

```
AddImage: PROC [
    h: Handle,
    box: Box,
    imageProps: ReadonlyImageProps,
    frameProps: ReadonlyFrameProps,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

```
ImageProps: TYPE = LONG POINTER TO ImagePropsRec;
```

```
ReadonlyImageProps: TYPE = LONG POINTER TO READONLY ImagePropsRec;
```

```
ImagePropsRec: TYPE = RECORD [
    name: XString.Reader,
    spare1: LONG CARDINAL];
```

AddImage adds an image frame to the specified graphics container.

```
AddTable: PROC [
    h: Handle,
    box: Box,
    table: DocInterchangeDefs.Instance,
    frameProps: ReadonlyFrameProps,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchangeDefs.Caption];
```

AddTable adds a table frame to the graphics container h. table should be the instance returned from TableInterchangeDefs.FinishTable. This operation is currently defined in GraphicsInterchangeExtra2Defs.

```
AddTextFrame: PROC [
    h: Handle,
    box: Box,
    frameProps: ReadonlyFrameProps,
    textFrameProps: ReadonlyTextFrameProps,
    wantTextHandle,
    wantTopCaptionHandle,
    wantBottomCaptionHandle,
    wantLeftCaptionHandle,
    wantRightCaptionHandle: BOOLEAN ← FALSE]
    RETURNS [
        text: TextInterchangeDefs.Text,
        topCaption, bottomCaption,
        leftCaption, rightCaption: DocInterchange.Caption];

TextFrameProps: TYPE = LONG POINTER TO TextFramePropsRec;

ReadonlyTextFrameProps: TYPE =
    LONG POINTER TO READONLY TextFramePropsRec;

TextFramePropsRec: TYPE = RECORD [
    expandRight, expandBottom, transparent: BOOL,
    tFrameProps: TextInterchangeDefs.TFramePropsRec,
    spare1: LONG CARDINAL];
```

AddTextFrame adds a text frame to the specified graphics container. If the client specifies wantTextHandle = TRUE, it will return a handle to a text frame. The handle may then be used to add text to the central area of the frame.

75.2.1.5 Adding to a cusp button

The following routines allow the client to add textual information to a cusp button program.

```
AppendCharToButtonProgram: PROC [
    to: ButtonProgram,
    char: XChar.Character,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];
```

Add a character to the button program. **nToAppend** is the number of copies of the character to be added; **fontProps** are the properties of the character.

```
AppendNewParagraphToButtonProgram: PROC [
    to: ButtonProgram,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];
```

Add a new paragraph character with specified properties to the button program.

```
AppendTextToButtonProgram: PROC [
    to: ButtonProgram,
    text: XString.Reader,
    textEndContext: XString.Context,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

Add a string with specified properties to button program. For efficiency, the client should include **textEndContext** if known. Do not append **newParagraph** characters [set: 0, code: 35B] with this operation--use **AppendNewParagraphToButtonProgram** for that.

75.2.1.6 Adding miscellaneous graphics

```
AddOther: PROC [
    h: Handle,
    box: Box,
    instance:DocInterchangeDefs.Instance];
```

AddOther is provided to allow addition of charts and other as yet undefined objects. For information on charts, see **ChartDataInstallDefs**.

75.2.1.7 Release routines

```
ReleaseButtonProgram: PROC [
    bpPtr: LONG POINTER TO ButtonProgram];
```

ReleaseButtonProgram releases the handles obtained from **AddButtonProgram**. Like Mesa's FREE operator, this routine take a pointer to the object to be freed, and sets the handle itself to NIL. Thus, after a call to **ReleaseButtonProgram**, **ButtonProgram** will be NIL.

75.2.1.8 Finish routines

When everything has been added to a graphics container, the client should call a **Finish** routine.

```
FinishButton: PROC [bfh: Handle];

FinishCluster: PROC [ch: Handle];

FinishGraphics: PROC [h: Handle]
    RETURNS [graphics: DocInterchangeDefs.Instance];

FinishGraphicsFrame: PROC [gfh: Handle];
```

bfh, **ch**, **h**, and **gfh** are the handles obtained from the corresponding **Start** routines. The client will typically pass the **DocInterchangeDefs.Instance** returned by **FinishGraphics** to **DocInterchangeDefs.AppendAnchoredFrame**.

75.2.2 Reading graphics

To read the contents of a graphics frame, the client should call **Enumerate**. **Enumerate** takes as parameters a graphics container and a list of call back procedures, one for each of the kinds of items that might be in the graphics container. **Enumerate** will proceed through the graphics container, calling the appropriate procedure for each item that it encounters.

Each enumeration procedure takes parameters that describe the properties of the object. These properties are temporary, and will be destroyed after the client procedure returns. If the client wishes to save any of these properties, it must explicitly copy them.

Client **EnumProcs** should not call any **Release*** routines on anything passed them as a parameter. The Enumerator always releases containers after calling each **EnumProc**.

In the case of a cusp button, cluster, or nested graphics frame, the client can recursively call **Enumerate** to get the contents of the nested frame. There are also related enumeraters, **TextInterchangeDefs.EnumerateText** and **EnumerateButtonProgram**, that enumerate the contents of a text frame and a cusp button, respectively.

75.2.2.1 Enumerate and its callbacks

```
Enumerate: PROC [
    doc: DocInterchangeDefs.Doc,
    graphicsContainer: DocInterchangeDefs.Instance,
    procs: EnumProcs,
    clientData: LONG POINTER ← NIL]
    RETURNS [dataSkipped: BOOLEAN];

EnumProcs: TYPE = LONG POINTER TO EnumProcsRecord;

EnumProcsRecord: TYPE = RECORD [
    bitmapProc: BitmapProc ← NIL,
    buttonProc: ButtonProc ← NIL,
    clusterProc: ClusterProc ← NIL,
    curveProc: CurveProc ← NIL,
    ellipseProc: EllipseProc ← NIL,
    formFieldProc: FormFieldProc ← NIL,
    frameProc: FrameProc ← NIL,
    imageProc: ImageProc ← NIL,
    lineProc: LineProc ← NIL,
    otherProc: OtherProc ← NIL,
    pointProc: PointProc ← NIL,
    rectangleProc: RectangleProc ← NIL,
    textFrameProc: TextFrameProc ← NIL,
    triangleProc: TriangleProc ← NIL];
```

To enumerate an anchored graphics frame as a graphics container, pass the anchoredFrame parameter of the DocInterchangeDefs.AnchoredFrameProc into Enumerate, not the content parameter. See the DocInterchangeDefs documentation for AnchoredFrameProc.

```
BitmapProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    bitmapProps: ReadonlyBitmapProps,
    frameProps: ReadonlyFrameProps]
    RETURNS [stop: BOOLEAN ← FALSE];

ButtonProc: TYPE = PROC [
    clientData: LONG POINTER,
    graphicsContainer:DocInterchangeDefs.Instance,
    box: Box,
    buttonProps: ReadonlyButtonProps,
    frameProps: ReadonlyFrameProps,
    buttonProgram: ButtonProgram]
    RETURNS [stop: BOOLEAN ← FALSE];

ClusterProc: TYPE = PROC [
    clientData: LONG POINTER,
    graphicsContainer: DocInterchangeDefs.Instance,
    box: Box]
    RETURNS [stop: BOOLEAN ← FALSE];

CurveProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    curveProps: ReadonlyCurveProps]
    RETURNS [stop: BOOLEAN ← FALSE];

EllipseProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    ellipseProps: ReadonlyEllipseProps]
    RETURNS [stop: BOOLEAN ← FALSE];

FormFieldProc: TYPE = PROC [
    clientData: LONG POINTER, box: Box,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    frameProps: ReadonlyFrameProps,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps,
    expandRight, expandBottom: BOOL,
    content: DocInterchangeDefs.Field]
    RETURNS [stop: BOOLEAN ← FALSE];

FrameProc: TYPE = PROC [
    clientData: LONG POINTER,
    graphicsContainer: DocInterchangeDefs.Instance,
    box: Box,
    frameProps: ReadonlyFrameProps,
    name, description: XString.Reader,
    spareProps: LONG POINTER]
    RETURNS [stop: BOOLEAN ← FALSE];

ImageProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    imageProps: ReadonlyImageProps,
    frameProps: ReadonlyFrameProps]
    RETURNS [stop: BOOLEAN ← FALSE];

LineProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    lineProps: ReadonlyLineProps]
    RETURNS [stop: BOOLEAN ← FALSE];

OtherProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    instance: DocInterchangeDefs.Instance,
    objectType: OtherObjectType]
    RETURNS [stop: BOOLEAN ← FALSE];

OtherObjectType: TYPE = MACHINE DEPENDENT {
    illusFrame(0), barchart, linechart, piechart, pieslice, table, equation,
    firstAvailable, lastAvailable(255)};
```

Although data driven charts are not yet fully supported by any Interchange interface, it is possible to enumerate the graphic components of one. In the **OtherProc**, simply call **Enumerate** on the **instance** when it is a chart, and the appropriate procedures will be called for its contained graphic objects. Think of it as a cluster.

```
PointProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    pointProps: ReadonlyPointProps]
    RETURNS [stop: BOOLEAN ← FALSE];

RectangleProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    rectangleProps: ReadonlyRectangleProps]
    RETURNS [stop: BOOLEAN ← FALSE];

TextFrameProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    frameProps: ReadonlyFrameProps,
    textFrameProps: ReadonlyTextFrameProps,
    content: TextInterchangeDefs.Text]
    RETURNS [stop: BOOLEAN ← FALSE];

TriangleProc: TYPE = PROC [
    clientData: LONG POINTER,
    box: Box,
    triangleProps: ReadonlyTriangleProps]
    RETURNS [stop: BOOLEAN ← FALSE];
```

75.2.2.2 Getting extra properties

```
GetExtraAnchoredFrameProps: PROC [
    doc: DocInterchangeDefs.Doc,
    anchoredFrame: DocInterchangeDefs.Instance,
    spareProps: LONG POINTER ← NIL,
    zone: UNCOUNTED ZONE]
    RETURNS [name, description: XString.ReaderBody];
```

The **name** and **description** properties can be retrieved from an anchored frame with **GetExtraAnchoredFrameProps**. **spareProps** and **zone** are for future use. The returned values are not allocated from zone and so are read-only; the client should not attempt to call **XString.FreeReaderBytes** on them. Within a **DocInterchangeDefs.AnchoredFrameProc** the client should pass the **anchoredFrame** parameter to **GetExtraAnchoredFrameProps**, rather than the **content** parameter.

```
GetExtraFormFieldProps: PROC [contentOfFormField: DocInterchangeDefs.Field]
    RETURNS [textFrameProps: ReadonlyTextFrameProps];
```

This operation returns a readonly pointer to the text frame properties of a form field. **GetExtraFormFieldProps** may only be called from within a **FormFieldProc**. The enumerator owns the storage of all the returned properties. **GetExtraFormFieldProps** is currently defined in **GraphicsInterchangeExtra1Defs**.

```
GetNestedTableProps: PROC [
    doc: DocInterchangeDefs.Doc,
	instance: DocInterchangeDefs.Instance,
	frameProps: FrameProps]
	RETURNS [content: DocInterchangeDefs.Instance];
```

GetNestedTableProps fills in **frameProps** with the frame props of the nested table frame and returns **content**. May be called only within an **OtherProc** during an enumeration, and the instance passed to **GetNestedTableProps** must be the same as the one passed to the **OtherProc**. The entries of the **frameProps.captionContent** are readonly and are owned by the enumerator, so they are valid only during that call to the **OtherProc**. Do not call **DocInterchangeDefs.ReleaseCaption** on any of them. Use **TableInterchangeDefs.EnumerateTable [content, ...]** to obtain the rest of the table's properties. **GetNestedTableProps** is currently defined in **GraphicsInterchangeExtra2Defs**.

```
GetPiesliceProps: PROC [
	doc: DocInterchangeDefs.Doc,
	instance: DocInterchangeDefs.Instance,
	piesliceProps: PiesliceProps];
```

GetPiesliceProps fills in **piesliceProps** with the properties of the pieslice. May be called only within an **OtherProc** when the object is a pieslice. **GetPiesliceProps** is currently defined in **GraphicsInterchangeExtra3Defs**.

75.2.2.3 Enumerating cusp button programs

```
EnumerateButtonProgram: PROC [
	buttonProgram: ButtonProgram,
	procs: ButtonProgramEnumProcs,
	clientData: LONG POINTER ← NIL]
	RETURNS [dataSkipped: BOOLEAN];

ButtonProgramEnumProcs: TYPE =
	LONG POINTER TO ButtonProgramEnumProcsRecord;

ButtonProgramEnumProcsRecord: TYPE = RECORD [
	newParagraphProc: DocInterchangeDefs.NewParagraphProc ← NIL,
	textProc: DocInterchangeDefs.TextProc ← NIL];
```

EnumerateButtonProgram enumerates the contents of buttonProgram, calling the client-supplied **procs** as appropriate. **clientData** is passed to each of the call-back procedures during enumeration.

75.2.3 Constants

```
nullBitmapProps: BitmapPropsRec = [
	opaque: TRUE,
	xOffset: 0,
	yOffset: 0,
	printFile: XString.nullReaderBody,
	displaySource: nullBmDisplay,
	scalingProps: nullBitmapScalingProps,
    spare1: 0];
    nullBmDisplay: BmDisplay = [internal[bm: NIL]];

nullBitmapScalingProps: BitmapScalingProps = [automatic[shape: similar]];

nullButtonProps: ButtonPropsRec = [name: NIL, spare1: 0];

nullCurveProps: CurvePropsRec = [
    brush: [0, solid],
    lineEndNW: flush,
    lineEndSE: flush,
    lineEndHeadNW: none,
    lineEndHeadSE: none,
    direction: WE,
    placeNW: [0, 0],
    placeApex: [0, 0],
    placeSE: [0, 0],
    placePeak: [0, 0],
    fixedAngle: FALSE,
    spare1: 0];

nullEccentricCurveProps: EccentricCurvePropsRec = [
    brush: [0, solid],
    lineEndNW: flush,
    lineEndSE: flush,
    lineEndHeadNW: none,
    lineEndHeadSE: none,
    direction: WE,
    placeNW: [0, 0],
    placeApex: [0, 0],
    placeSE: [0, 0],
    eccentricity: 0,
    fixedAngle: FALSE,
    spare1: 0];

nullEllipseProps: EllipsePropsRec = [
    brush: [0, solid],
    shading: [none, ALL[FALSE]],
    fixedShape: FALSE,
    spare1: 0];

nullFrameProps: FramePropsRec = [
    brush: [0, solid],
    fixedShape: FALSE,
    margins: ALL[0],
    captionContent: ALL[NIL],
    spare1: 0];

nullImageProps: ImagePropsRec = [name: NIL, spare1: 0];

nullLineProps: LinePropsRec = [
    brush: [0, solid],
    lineEndNW: flush,
    lineEndSE: flush,
    lineEndHeadNW: none,
    lineEndHeadSE: none,
    direction: WE,
    fixedAngle: FALSE,
    spare1: 0];

nullPiesliceProps: PieslicePropsRec = [
    brush: [2, solid],
    shading: [none, ALL[FALSE]],
    center: [0, 0],
    start: [0, 0],
    stop: [0, 0],
    spare1: 0];
```

nullPiesliceProps is currently defined in GraphicsInterchangeExtra3Defs.

```
nullPointProps: PointPropsRec = [
    wthbrush: 0,
    pointStyle: round,
    pointFill: solid,
    spare1: 0];

nullRectangleProps: RectanglePropsRec = [
    brush: [0, solid],
    shading: [none, ALL[FALSE]],
    fixedShape: FALSE,
    spare1: 0];

nullTextFrameProps: TextFramePropsRec = [
    expandRight: FALSE,
    expandBottom: FALSE,
    transparent: FALSE,
    tFrameProps: TextInterchangeDefs.nullTFrameProps,
    spare1: 0];

nullTriangleProps: TrianglePropsRec = [
    brush: [0, solid],
    shading: [none, ALL[FALSE]],
    place1: [0, 0],
    place2: [0, 0],
    place3: [0, 0],
    fixedShape: FALSE,
    spare1: 0];
```

75.3 Usage/Examples

The following code copies anchored graphic and cusp button frames from a source document being enumerated to a target document being created. What follows is not a complete program; the jumping off point is a DocInterchangeDefs.AnchoredFrameProc. This code fits in with the example code listed in the **DocInterchangeDefs** documentation. Some of the declarations below are copied from the **DocInterchangeDefs** example.

```
-- Types

CaptionsHandle: TYPE = LONG POINTER TO CaptionsRec;
CaptionsRec: TYPE = RECORD [tc, bc, lc, rc: DocInterchangeDefs.Caption];

DICtxtHandle: TYPE = LONG POINTER TO DICtxt;
DICtxt: TYPE = RECORD [
   sourceDoc, targetDoc: DocInterchangeDefs.Doc,
   ignoreNewPar, ignorePFC, aborted, error: BOOLEAN];
<< A DICtxtHandle is passed as clientData to procs called by
   DocInterchangeDefs.Enumerate. >>

GICtxtHandle: TYPE = LONG POINTER TO GICtxt;
GICtxt: TYPE = RECORD [
   h: GraphicsInterchangeDefs.Handle,
   sourceDoc, targetDoc: DocInterchangeDefs.Doc,
   error: BOOL];
<< A GICtxtHandle is passed as clientData to procs called by
      GraphicsInterchangeDefs.Enumerate. >>

-- Constants

z: UNCOUNTED ZONE = BWSZone.shortLifetime;

-- Variables

giEnumProcsRec: GraphicsInterchangeDefs.EnumProcsRecord ← [
   bitmapProc: Bitmap, buttonProc: Button, clusterProc: Cluster, curveProc: Curve,
   ellipseProc: Ellipse, formFieldProc: FormField, frameProc: Frame, imageProc: Image,
   lineProc: Line, otherProc: Other, pointProc: Point, rectangleProc: Rectangle,
   textFrameProc: TextFrame, triangleProc: Triangle];
giEnumProcs: GraphicsInterchangeDefs.EnumProcs = @giEnumProcsRec;

<< Called when an anchored frame was encountered in the source document.
   Copies the frame and its contents to the target document. >>
AppendAnchoredFrameToTargetDoc: DocInterchangeDefs.AnchoredFrameProc = {
   diCtxt: DICtxtHandle = clientData;
   sourceCaptions: CaptionsRec ← [
      topCaption, bottomCaption, leftCaption, rightCaption];
   newCaptions: CaptionsRec;
   IF (diCtxt.aborted ← BackgroundProcess.UserAbort[]) THEN {
      BackgroundProcess.ResetUserAbort[]; RETURN[stop: TRUE]; };
   SELECT type FROM
      graphics => {
         h: GraphicsInterchangeDefs.Handle ← GraphicsInterchangeDefs.StartGraphics[
            diCtxt.targetDoc];
         giCtxt: GICtxt ← [h, diCtxt.sourceDoc, diCtxt.targetDoc, FALSE];
         gc: DocInterchangeDefs.Instance;
         [] ← GraphicsInterchangeDefs.Enumerate[
            diCtxt.sourceDoc, anchoredFrame, giEnumProcs, @giCtxt];
         {
         nameRB, descriptionRB: XString.ReaderBody;
         gc ← GraphicsInterchangeDefs.FinishGraphics[h];
```

```
        [nameRB, descriptionRB] ←
           GraphicsInterchangeDefs.GetExtraAnchoredFrameProps[
           doc: diCtxt.sourceDoc, anchoredFrame: anchoredFrame, zone: NIL];
        GraphicsInterchangeDefs.SetExtraAnchoredFrameProps[
           doc: diCtxt.targetDoc, anchoredFrame: gc, name: @nameRB,
           description: @descriptionRB,
           selections: [name: TRUE, description: TRUE, spareProps: FALSE]]};
        [, newCaptions.tc, newCaptions.bc, newCaptions.lc, newCaptions.rc] ←
           DocInterchangeDefs.AppendAnchoredFrame[
           to: diCtxt.targetDoc, type: graphics,
           anchoredFrameProps: anchoredFrameProps,
           content: gc, wantTopCaptionHandle: topCaption # NIL,
           wantBottomCaptionHandle: bottomCaption # NIL,
           wantLeftCaptionHandle: leftCaption # NIL,
           wantRightCaptionHandle: rightCaption # NIL,
           anchorFontProps: anchorFontProps !
           DocInterchangeDefs.Error => {
                      diCtxt.error ← TRUE; stop ← TRUE; CONTINUE}];
        IF ~diCtxt.error THEN FillInFrameCaptions[@sourceCaptions, @newCaptions];
        }; --graphics
     cuspButton => {
        bProps: GraphicsInterchangeDefs.ButtonPropsRec;
        bProgSource: GraphicsInterchangeDefs.ButtonProgram ←
           ButtonInterchangeDefs.ButtonInfoForAnchoredFrame[
           diCtxt.sourceDoc, anchoredFrame, @bProps, z];
        bProgTarget: GraphicsInterchangeDefs.ButtonProgram;
        h: GraphicsInterchangeDefs.Handle;
        giCtxt: GICtxt;
        bProcs: GraphicsInterchangeDefs.ButtonProgramEnumProcsRecord ← [
           newParagraphProc: AppendNewParToProg, textProc: AppendTextToProg];

        AppendNewParToProg: DocInterchangeDefs.NewParagraphProc = {
           GraphicsInterchangeDefs.AppendNewParagraphToButtonProgram[
              bProgTarget, paraProps, fontProps];
           }; --AppendNewParToProg

        AppendTextToProg: DocInterchangeDefs.TextProc = {
           GraphicsInterchangeDefs.AppendTextToButtonProgram[
              bProgTarget, text, textEndContext, fontProps];
           }; --AppendTextToProg

        [h, bProgTarget] ← ButtonInterchangeDefs.StartButton[
           diCtxt.targetDoc, @bProps, (bProgSource # NIL)];
        [] ← GraphicsInterchangeDefs.EnumerateButtonProgram[
           bProgSource, @bProcs];
        GraphicsInterchangeDefs.ReleaseButtonProgram[@bProgTarget];
        ButtonInterchangeDefs.ReleaseReadOnlyButtonProgram[@bProgSource];
        giCtxt ← [h, diCtxt.sourceDoc, diCtxt.targetDoc, FALSE];
        [] ← GraphicsInterchangeDefs.Enumerate[
           diCtxt.sourceDoc, anchoredFrame, giEnumProcs, @giCtxt];
        [, newCaptions.tc, newCaptions.bc, newCaptions.lc, newCaptions.rc] ←
           DocInterchangeDefs.AppendAnchoredFrame[
           to: diCtxt.targetDoc, type: cuspButton,
           anchoredFrameProps: anchoredFrameProps,
           content: ButtonInterchangeDefs.FinishButton[h],
           wantTopCaptionHandle: topCaption # NIL,
            wantBottomCaptionHandle: bottomCaption # NIL,
            wantLeftCaptionHandle: leftCaption # NIL,
            wantRightCaptionHandle: rightCaption # NIL,
            anchorFontProps: anchorFontProps !
            DocInterchangeDefs.Error = > {
              diCtxt.error ← TRUE; stop ← TRUE; CONTINUE}];
          IF ~diCtxt.error THEN FillInFrameCaptions[@sourceCaptions, @newCaptions];
          };
        ENDCASE;
    }; -- AppendAnchoredFrame

  << Called when a bitmap frame was encountered while enumerating the contents
     of a graphics container. >>
  Bitmap: GraphicsInterchangeDefs.BitmapProc = {
    ctxt: GICtxtHandle = clientData;
    sourceCaptions: CaptionsRec ← [
      frameProps.captionContent[top], frameProps.captionContent[bottom],
      frameProps.captionContent[left], frameProps.captionContent[right]];
    newCaptions: CaptionsRec;
    [newCaptions.tc, newCaptions.bc, newCaptions.lc, newCaptions.rc] ←
          GraphicsInterchangeDefs.AddBitmap[
      ctxt.h, box, bitmapProps, frameProps, sourceCaptions.tc # NIL,
      sourceCaptions.bc # NIL, sourceCaptions.lc # NIL, sourceCaptions.rc # NIL];
    FillInFrameCaptions[@sourceCaptions, @newCaptions];
    }; -- Bitmap

  << Called when a Cusp button was encountered while enumerating the contents
     of a graphics container. >>
  Button: GraphicsInterchangeDefs.ButtonProc = {
    ctxt: GICtxtHandle = clientData;
    bfh: GraphicsInterchangeDefs.Handle;
    newProgram: GraphicsInterchangeDefs.ButtonProgram;
    bpProcs: GraphicsInterchangeDefs.ButtonProgramEnumProcsRecord ← [
      newParagraphProc: AppendNewParToProgram,
      textProc: AppendTextToProgram];
    newCtxt: GICtxt;
    sourceCaptions: CaptionsRec ← [
      frameProps.captionContent[top], frameProps.captionContent[bottom],
      frameProps.captionContent[left], frameProps.captionContent[right]];
    newCaptions: CaptionsRec;

    AppendNewParToProgram: DocInterchangeDefs.NewParagraphProc = {
      GraphicsInterchangeDefs.AppendNewParagraphToButtonProgram[
        newProgram, paraProps, fontProps];
      }; -- AppendNewParToProgram

    AppendTextToProgram: DocInterchangeDefs.TextProc = {
      GraphicsInterchangeDefs.AppendTextToButtonProgram[
        newProgram, text, textEndContext, fontProps];
      }; -- AppendTextToProgram

    [bfh, newProgram, newCaptions.tc, newCaptions.bc, newCaptions.lc,
      newCaptions.rc] ←
      GraphicsInterchangeDefs.StartButton[
      ctxt.h, box, buttonProps, frameProps, buttonProgram # NIL,
      sourceCaptions.tc # NIL, sourceCaptions.bc # NIL, sourceCaptions.lc # NIL,
      sourceCaptions.rc # NIL];
   newCtxt ← [bfh, ctxt.sourceDoc, ctxt.targetDoc, FALSE];
   [] ← GraphicsInterchangeDefs.Enumerate[
      ctxt.sourceDoc, graphicsContainer, giEnumProcs, @newCtxt];
   GraphicsInterchangeDefs.FinishButton[bfh];
   [] ← GraphicsInterchangeDefs.EnumerateButtonProgram[
      buttonProgram, @bpProcs];
   GraphicsInterchangeDefs.ReleaseButtonProgram[@newProgram];
   FillInFrameCaptions[@sourceCaptions, @newCaptions];
   }; -- Button

<< Called when a cluster was encountered while enumerating the contents
   of a graphics container. >>
Cluster: GraphicsInterchangeDefs.ClusterProc = {
   ctxt: GICtxtHandle = clientData;
   ch: GraphicsInterchangeDefs.Handle =
      GraphicsInterchangeDefs.StartCluster[ctxt.h, box];
   newCtxt: GICtxt ← [ch, ctxt.sourceDoc, ctxt.targetDoc, FALSE];
   [] ← GraphicsInterchangeDefs.Enumerate[
      ctxt.sourceDoc, graphicsContainer, giEnumProcs, @newCtxt];
   GraphicsInterchangeDefs.FinishCluster[ch];
   }; -- Cluster

CopyColumnInfo: PROC [
   source: TableInterchangeDefs.ColumnInfo, zone: UNCOUNTED ZONE]
   RETURNS [new: TableInterchangeDefs.ColumnInfo] = {
   new ← zone.NEW[TableInterchangeDefs .ColumnInfoSeq[source.length]];
   FOR i: CARDINAL IN [0..source.length) DO
      new[i] ← source[i];
      new[i].headerEntryRec.content ← [
         write[FillInText, @source[i].headerEntryRec.content]];
      IF source[i].subcolumnInfo # NIL THEN
         new[i].subcolumnInfo ← CopyColumnInfo[source[i].subcolumnInfo, zone];
      ENDLOOP;
   }; -- CopyColumnInfo

CopyRowContentInfo: PROC [
   source: TableInterchangeDefs.RowContent, zone: UNCOUNTED ZONE]
   RETURNS [new: TableInterchangeDefs.RowContent] = {
   new ← zone.NEW[TableInterchangeDefs .RowContentSeq[source.length]];
   new.topMargin ← source.topMargin;
   new.bottomMargin ← source.bottomMargin;
   new.line ← source.line;
   new.verticalAlignment ← source.verticalAlignment;
   FOR i: CARDINAL IN [0..source.length) DO
      new.rowdata[i] ← source.rowdata[i];
      new.rowdata[i].content ← [write[FillInText, @source[i].content]];
      IF source.rowdata[i].subRows # NIL THEN {
         new.rowdata[i].subRows ← zone.NEW[
            TableInterchangeDefs .SubRowsRec[source.rowdata[i].subRows.length]];
         new.rowdata[i].subRows.length ← source.rowdata[i].subRows.length;
         FOR j: CARDINAL IN [0..source.rowdata[i].subRows.length) DO
            new.rowdata[i].subRows.rows[j] ← CopyRowContentInfo[
               source.rowdata[i].subRows.rows[j], zone];
            ENDLOOP;
         }; -- had subrows
   ENDLOOP;
   }; -- CopyRowContentInfo

<< Called when a curve was encountered while enumerating the contents
   of a graphics container. >>
Curve: GraphicsInterchangeDefs.CurveProc = {
   ctxt: GICtxtHandle = clientData;
   GraphicsInterchangeDefs.AddCurve[ctxt.h, box, curveProps]};

<< Called when an ellipse was encountered while enumerating the contents
   of a graphics container. >>
Ellipse: GraphicsInterchangeDefs.EllipseProc = {
   ctxt: GICtxtHandle = clientData;
   GraphicsInterchangeDefs.AddEllipse[ctxt.h, box, ellipseProps]};

<< Called by any proc that's appending a frame to the target doc,
   FillInFrameCaptions enumerates the contents of each source caption and
   appends everything found to the corresponding new caption. >>
FillInFrameCaptions: PROC [sourceCaptions, targetCaptions: CaptionsHandle] = {
   crntTC: DocInterchangeDefs.TextContainer;
   << The current TextContainer to append stuff to. >>
   procs: DocInterchangeDefs.EnumProcsRecord ← [
      fieldProc: AppendFieldToCrntTC, newParagraphProc: AppendNewParToCrntTC,
      textProc: AppendTextToCrntTC];

   AppendNewParToCrntTC: DocInterchangeDefs.NewParagraphProc = {
      DocInterchangeDefs.AppendNewParagraph[crntTC, paraProps, fontProps]};

   AppendTextToCrntTC: DocInterchangeDefs.TextProc = {
      DocInterchangeDefs.AppendText[crntTC, text, textEndContext, fontProps]};

   AppendFieldToCrntTC: DocInterchangeDefs.FieldProc = {
      procs: DocInterchangeDefs.EnumProcsRecord ← [
         newParagraphProc: AppendNewParToCrntTC,
         textProc: AppendTextToCrntTC];
      newField: DocInterchangeDefs.Field ← DocInterchangeDefs.AppendField[
         crntTC, fieldProps, fontProps];
      saveTC: DocInterchangeDefs.TextContainer = crntTC;
      crntTC ← [field[h: newField]];
      [] ← DocInterchangeDefs.Enumerate[[field[h: field]], @procs];
      DocInterchangeDefs.ReleaseField[@newField];
      crntTC ← saveTC;
      }; -- AppendFieldToCrntTC

   -- start of FillInFrameCaptions
   crntTC ← [caption[h: targetCaptions.tc]];
   [] ← DocInterchangeDefs.Enumerate[[caption[h: sourceCaptions.tc]], @procs];
   DocInterchangeDefs.ReleaseCaption[@targetCaptions.tc];

   crntTC ← [caption[h: targetCaptions.bc]];
   [] ← DocInterchangeDefs.Enumerate[[caption[h: sourceCaptions.bc]], @procs];
   DocInterchangeDefs.ReleaseCaption[@targetCaptions.bc];

   crntTC ← [caption[h: targetCaptions.lc]];
   [] ← DocInterchangeDefs.Enumerate[[caption[h: sourceCaptions.lc]], @procs];
   DocInterchangeDefs.ReleaseCaption[@targetCaptions.lc];
```

```
crntTC ← [caption[h: targetCaptions.rc]];
[] ← DocInterchangeDefs.Enumerate[[caption[h: sourceCaptions.rc]], @procs];
DocInterchangeDefs.ReleaseCaption[@targetCaptions.rc];
}; -- FillInFrameCaptions

FillInText: TableInterchangeDefs.FillInTextProc = {
  sourceEntryContent: LONG POINTER TO TableInterchangeDefs.EntryContent =
     clientData;
  sourceText: TextInterchangeDefs.Text;
  targetText: TextInterchangeDefs.Text = text;
  procs: TextInterchangeDefs.TextEnumProcsRecord ← [
  newParagraphProc: HitNP, textProc: HitText];

  HitNP: DocInterchangeDefs.NewParagraphProc = {
  TextInterchangeDefs.AppendNewParagraphToText[
     targetText, paraProps, fontProps]};

  HitText: DocInterchangeDefs.TextProc = {
  TextInterchangeDefs.AppendTextToText[
     targetText, text, textEndContext, fontProps]};

  -- start of FillInText
  WITH c: sourceEntryContent↑ SELECT FROM
     read => sourceText ← c.obtainTextProc[c.obtainTextData];
     write => ERROR;
     ENDCASE;
  [] ← TextInterchangeDefs.EnumerateText[sourceText, @procs];
  TextInterchangeDefs.ReleaseText[@sourceText];
  }; -- FillInText

<< Proc that's called for each form-field frame in the graphic container. >>
FormField: GraphicsInterchangeDefs.FormFieldProc = {
  ctxt: GICtxtHandle = clientData;
  field: DocInterchangeDefs.Field;
  sourceCaptions: CaptionsRec ← [
     frameProps.captionContent[top], frameProps.captionContent[bottom],
     frameProps.captionContent[left], frameProps.captionContent[right]];
  newCaptions: CaptionsRec;
  procs: DocInterchangeDefs.EnumProcsRecord ← [
     newParagraphProc: AppendNewParToField,
     textProc: AppendTextToField];

  AppendNewParToField: DocInterchangeDefs.NewParagraphProc = {
     DocInterchangeDefs.AppendNewParagraph[
        [field[h: field]], paraProps, fontProps]};

  AppendTextToField: DocInterchangeDefs.TextProc = {
     DocInterchangeDefs.AppendText[
        [field[h: field]], text, textEndContext, fontProps]};

  [field, newCaptions.tc, newCaptions.bc, newCaptions.lc, newCaptions.rc] ←
     GraphicsInterchangeExtra1Defs.AddFormFieldX[
     ctxt.h, box, fieldProps, frameProps,
     GraphicsInterchangeExtra1Defs.GetExtraFormFieldProps[content], paraProps,
     fontProps, content # NIL, sourceCaptions.tc # NIL, sourceCaptions.bc # NIL,
      sourceCaptions.lc # NIL, sourceCaptions.rc # NIL];
   [] ← DocInterchangeDefs.Enumerate[
      [field[h: content]], @procs ! DocInterchangeDefs.Error => {...}];
   DocInterchangeDefs.ReleaseField[@field];
   FillInFrameCaptions[@sourceCaptions, @newCaptions];
   }; -- FormField

<< Called when a graphics frame was encountered while enumerating the contents
   of a graphics container. >>
Frame: GraphicsInterchangeDefs.FrameProc = {
   ctxt: GICtxtHandle = clientData;
   gfh: GraphicsInterchangeDefs.Handle;
   sourceCaptions: CaptionsRec ← [
      frameProps.captionContent[top], frameProps.captionContent[bottom],
      frameProps.captionContent[left], frameProps.captionContent[right]];
   newCaptions: CaptionsRec;
   newCtxt: GICtxt;
   [gfh, newCaptions.tc, newCaptions.bc, newCaptions.lc, newCaptions.rc] ←
      GraphicsInterchangeDefs.StartGraphicsFrame[
      ctxt.h, box, frameProps, name, description, NIL, sourceCaptions.tc # NIL,
      sourceCaptions.bc # NIL, sourceCaptions.lc # NIL, sourceCaptions.rc # NIL];
   newCtxt ← [gfh, ctxt.sourceDoc, ctxt.targetDoc, FALSE];
   [] ← GraphicsInterchangeDefs.Enumerate[
      ctxt.sourceDoc, graphicsContainer, giEnumProcs, @newCtxt];
   FillInFrameCaptions[@sourceCaptions, @newCaptions];
   }; -- Frame

FreeColumnInfo: PROC [
   c: LONG POINTER TO TableInterchangeDefs.ColumnInfo, zone: UNCOUNTED ZONE] = {
   FOR i: CARDINAL IN [0..c.length) DO
      IF c[i].subcolumnInfo # NIL THEN FreeColumnInfo[@c[i].subcolumnInfo, zone];
      ENDLOOP;
   zone.FREE[c];
   }; -- FreeColumnInfo

FreeRowContentInfo: PROC [
   r: LONG POINTER TO TableInterchangeDefs.RowContent, zone: UNCOUNTED ZONE] = {
   FOR i: CARDINAL IN [0..r.length) DO
      IF r.rowdata[i].subRows # NIL THEN
         FOR j: CARDINAL IN [0..r.rowdata[i].subRows.length) DO
            FreeRowContentInfo[@r.rowdata[i].subRows.rows[j], zone];
            ENDLOOP;
      ENDLOOP;
   zone.FREE[r];
   }; -- FreeRowContentInfo

<< Called when an image frame was encountered while enumerating the contents
   of a graphics container. >>
Image: GraphicsInterchangeDefs.ImageProc = {
   ctxt: GICtxtHandle = clientData;
   sourceCaptions: CaptionsRec ← [
      frameProps.captionContent[top], frameProps.captionContent[bottom],
      frameProps.captionContent[left], frameProps.captionContent[right]];
   newCaptions: CaptionsRec;
   [newCaptions.tc, newCaptions.bc, newCaptions.lc, newCaptions.rc] ←
      GraphicsInterchangeDefs.AddImage[
        ctxt.h, box, imageProps, frameProps, sourceCaptions.tc # NIL,
        sourceCaptions.bc # NIL, sourceCaptions.lc # NIL, sourceCaptions.rc # NIL];
      FillInFrameCaptions[@sourceCaptions, @newCaptions];
      }; -- Image

  << Called when a line was encountered while enumerating the contents
     of a graphics container. >>
  Line: GraphicsInterchangeDefs.LineProc = {
      ctxt: GICtxtHandle = clientData;
      GraphicsInterchangeDefs.AddLine[ctxt.h, box, lineProps]};

  Other: GraphicsInterchangeDefs.OtherProc = {
      ctxt: GICtxtHandle = clientData;
      SELECT objectType FROM
        barchart, linechart, piechart => { -- for now, turn these into a cluster
          ch: GraphicsInterchangeDefs.Handle = GraphicsInterchangeDefs.StartCluster[
            ctxt.h, box];
          newCtxt: GICtxt ← [ch, ctxt.sourceDoc, ctxt.targetDoc, FALSE];
          [] ← GraphicsInterchangeDefs.Enumerate[
            ctxt.sourceDoc, instance, giEnumProcs, @newCtxt];
          GraphicsInterchangeDefs.FinishCluster[ch];
          }; -- barchart, linechart, piechart
        pieslice => {
          piesliceProps: GraphicsInterchangeExtra3Defs.PiesliceProps Rec;
                    GraphicsInterchangeExtra3Defs.GetPiesliceProps[
            ctxt.sourceDoc, instance, @piesliceProps];
          GraphicsInterchangeExtra3Defs.AddPieslice[ctxt.h, box, @piesliceProps];
          }; -- pieslice
        table => {
          sourceCaptions, newCaptions: CaptionsRec;
          tableEnumProcs: TableInterchangeDefs.EnumProcsRec ← [
            TableProc, ColumnsProc, RowProc];
          tableProps: TableInterchangeDefs.TablePropsRec;
          extraTableProps: TableInterchangeExtra1Defs.ExtraTablePropsRec;
          h: TableInterchangeDefs.Handle ← NIL;
          frameProps: GraphicsInterchangeDefs.FramePropsRec;
          content: DocInterchangeDefs.Instance;

          TableProc: TableInterchangeDefs.TableProc = {
            tableProps ← props↑;
            IF props.spare1 # 0 THEN {
              tableProps.spare1 ← LOOPHOLE[LONG[@extraTableProps]];
              extraTableProps.deferOnPaginate ← LOOPHOLE[props.spare1,
                TableInterchangeExtra1Defs.ExtraTableProps].deferOnPaginate;
              extraTableProps.spare1 ← 0;
              };
            }; -- TableProc

          ColumnsProc: TableInterchangeDefs.ColumnsProc = {
            newColumnInfo: TableInterchangeDefs.ColumnInfo ←
              CopyColumnInfo[columns, z];
            h ← TableInterchangeDefs.StartTable[
              ctxt.targetDoc, @tableProps, newColumnInfo];
            FreeColumnInfo[@newColumnInfo, z];
            }; -- ColumnsProc
            RowProc: TableInterchangeDefs.RowProc = {
              newContent: TableInterchangeDefs.RowContent;
              IF BackgroundProcess.UserAbort[] THEN RETURN[stop: TRUE];
              newContent ← CopyRowContentInfo[content, z];
              TableInterchangeDefs.AppendRow[h, newContent];
              FreeRowContentInfo[@newContent, z];
              }; -- RowProc

            -- start of table arm of Other
            content ← GraphicsInterchangeExtra2Defs.GetNestedTableProps[
              ctxt.sourceDoc, instance, @frameProps];
            sourceCaptions ← [
              frameProps.captionContent[top], frameProps.captionContent[bottom],
              frameProps.captionContent[left], frameProps.captionContent[right]];
            TableInterchangeDefs.EnumerateTable[content, @tableEnumProcs];
            [newCaptions.tc, newCaptions.bc, newCaptions.lc, newCaptions.rc] ←
              GraphicsInterchangeExtra2Defs.AddTable[
              h: ctxt.h, box: box, table: TableInterchangeDefs.FinishTable[h].table,
              frameProps: @frameProps,
              wantTopCaptionHandle: frameProps.captionContent[top] # NIL,
              wantBottomCaptionHandle: frameProps.captionContent[bottom] # NIL,
              wantLeftCaptionHandle: frameProps.captionContent[left] # NIL,
              wantRightCaptionHandle: frameProps.captionContent[right] # NIL];
            FillInFrameCaptions[@sourceCaptions, @newCaptions];
            }; -- table
          ENDCASE;
      }; -- Other

    << Called when a point was encountered while enumerating the contents
       of a graphics container. >>
    Point: GraphicsInterchangeDefs.PointProc = {
      ctxt: GICtxtHandle = clientData;
      GraphicsInterchangeDefs.AddPoint[ctxt.h, box, pointProps]};

    << Called when a rectangle was encountered while enumerating the contents
       of a graphics container. >>
    Rectangle: GraphicsInterchangeDefs.RectangleProc = {
      ctxt: GICtxtHandle = clientData;
      GraphicsInterchangeDefs.AddRectangle[ctxt.h, box, rectangleProps]};

    << Called when a text frame was encountered while enumerating the contents
       of a graphics container. >>
    TextFrame: GraphicsInterchangeDefs.TextFrameProc = {
      ctxt: GICtxtHandle = clientData;
      sourceCaptions: CaptionsRec ← [
        frameProps.captionContent[top], frameProps.captionContent[bottom],
        frameProps.captionContent[left], frameProps.captionContent[right]];
      newCaptions: CaptionsRec;
      newContent: TextInterchangeDefs.Text;
      textProcs: TextInterchangeDefs.TextEnumProcsRecord ← [
        fieldProc: AppendFieldToContent,
        newParagraphProc: AppendNewParToContent,
        textProc: AppendTextToContent];

      AppendNewParToContent: DocInterchangeDefs.NewParagraphProc = {
        TextInterchangeDefs.AppendNewParagraphToText[
            newContent, paraProps, fontProps]};

      AppendTextToContent: DocInterchangeDefs.TextProc = {
         TextInterchangeDefs.AppendTextToText[
            newContent, text, textEndContext, fontProps]};

      AppendFieldToContent: DocInterchangeDefs.FieldProc = {
         procs: DocInterchangeDefs.EnumProcsRecord ← [
            newParagraphProc: AppendNewParToField, textProc: AppendTextToField];
         newField: DocInterchangeDefs.Field ← TextInterchangeDefs.AppendFieldToText[
            newContent, fieldProps, fontProps];

         AppendNewParToField: DocInterchangeDefs.NewParagraphProc = {
            DocInterchangeDefs.AppendNewParagraph[
               [field[h: newField]], paraProps, fontProps]};

         AppendTextToField: DocInterchangeDefs.TextProc = {
            DocInterchangeDefs.AppendText[
               [field[h: newField]], text, textEndContext, fontProps]};

         -- start of AppendFieldToContent
         [] ← DocInterchangeDefs.Enumerate[[field[h: field]], @procs];
         DocInterchangeDefs.ReleaseField[@newField]};

      [newContent, newCaptions.tc, newCaptions.bc, newCaptions.lc, newCaptions.rc] ←
         GraphicsInterchangeDefs.AddTextFrame[
         ctxt.h, box, frameProps, textFrameProps, content # NIL, sourceCaptions.tc # NIL,
         sourceCaptions.bc # NIL, sourceCaptions.lc # NIL, sourceCaptions.rc # NIL];
      [] ← TextInterchangeDefs.EnumerateText[content, @textProcs];
      TextInterchangeDefs.ReleaseText[@newContent];
      FillInFrameCaptions[@sourceCaptions, @newCaptions];
      }; -- TextFrame

   << Called when a triangle was encountered while enumerating the contents
      of a graphics container. >>
   Triangle: GraphicsInterchangeDefs.TriangleProc = {
      ctxt: GICtxtHandle = clientData;
      GraphicsInterchangeDefs.AddTriangle[ctxt.h, box, triangleProps]};
```

75.4 Index of Interface Items

76

IllustratorInterchangeDefs

76.1 Overview

The **IllustratorInterchangeDefs** interface provides the mechanisms for the creation and enumeration of frames within ViewPoint documents which contain Xerox Pro Illustrator graphics. It should be used in conjunction with the **DocInterchangeDefs** interface.

A Xerox Pro Illustrator frame is composed of a collection of forms. These forms may be simple graphical objects such as lines or curves, or may be more complex, such as clusters of forms, or text frames. Forms themselves can have various properties. For example, a line has a width and a color, and an area has a texture and a transparency.

76.1.1 Creating Pro Illustrator graphics

The client creates a Pro Illustrator frame within a document by calling **StartIllustrator**, passing in the required dimensions and frame parameters. This procedure returns a frame handle, which is used in subsequent operations on that frame.

Forms are created within a frame by calling various **Create*** procedures, such as **CreateLine**. These procedures take only the basic parameters to determine the placement of the form within the frame, along with a set of default properties, which determine the initial characteristics of the form. Subsequent modification of form properties can be accomplished by calling the appropriate **Set*Props** procedures.

Once created within the frame, a form must be posted in order for it to be displayed in the resulting Pro Illustrator frame. This is accomplished via **PostFormAtTop**, which causes the form to be added on (visual and structural) top of all forms previously posted in the same frame.

Transformations may be applied to a form after its creation. Procedures such as **Rotate** and **Shear** perform the corresponding transformations, relative to a pinned point supplied by the client.

Text frames within a Pro Illustrator frame are created by calling **CreateText***. The various procedures provided allow several different sources to be used for the text itself.

When the client has finished creating and manipulating forms, the frame is completed by calling **FinishIllustrator**. This procedure returns a **DocInterchangeDefs.Instance**, which should then be passed to **DocInterchangeDefs.AppendAnchoredFrame** to enter the Pro Illustrator frame into the document.

76.1.2 Reading Pro Illustrator graphics

The client can process the contents of an existing Pro Illustrator frame by calling **Enumerate**, passing in the frame handle and a procedure to be called back for each form. The frame handle itself can be obtained from an anchored frame by calling **GetFrameHandle**, passing in the **DocInterchangeDefs.Instance** which corresponds to the Pro Illustrator frame.

Basic information for each type of form can be obtained by calling **Get*Data**. This includes obtaining the location of the end points of a line, or the text content from a text frame. Particular groups of properties, not specific to any one type of form, can be obtained by calling **Get*Props**.

76.2 Interface Items

76.2.1 Creating Pro Illustrator graphics

76.2.1.1 Creating a frame

The following procedure creates a new Pro Illustrator frame in which graphics may subsequently be created.

```
StartIllustrator: PROCEDURE [
    doc: DocInterchangeDefs.Doc,
    lowerLeft, upperRight: Locn,
    frameUnit: FrameUnit,
    brush: FrameBrush,
    leftMargin, rightMargin, topMargin, bottomMargin: CARDINAL ← 0]
    RETURNS [
        ok: BOOLEAN,
        frame: FrameHandle];
```

StartIllustrator creates a new Pro Illustrator frame within **doc**. The size of the frame, exclusive of any frame margins, is determined by **lowerLeft** and **upperRight**, measured in **frameUnit** units. These values also determine the coordinate system to be used when forms are subsequently added to the frame (including the positive direction of the x and y axes), **frameUnit** defining the unit of measurement for subsequent operations within the frame.

```
Locn: TYPE = RECORD [x: Scalar, y: Scalar];
Scalar: TYPE = REAL;
```

```
FrameUnit: TYPE = {mica, millimeter, centimeter, point, point72, pica, inch};
```

Note: In the current implementation, **point** and **point72** are treated as equivalent, and correspond to exactly 72 points per inch.

The appearance of the frame border is determined by **brush**.

```
FrameBrush: TYPE = RECORD [
    frameBorderWidth: Pixels,
    frameBorderStyle: FrameBorderStyle];

Pixels: TYPE = CARDINAL;

FrameBorderStyle: TYPE = MACHINE DEPENDENT {
    invisible(0), solid(1), dashed(2), dotted(3), double(4), broken(5), (15)};
```

The size of the margin on each side of the frame is determined by **leftMargin**, **rightMargin**, **topMargin**, and **bottomMargin**, measured in micas.

The **FrameHandle** returned by **StartIllustrator** is passed to the various procedures used to add forms to the newly created frame.

```
FrameHandle: TYPE [2];
```

A frame may also be created by copying an existing Pro Illustrator frame.

```
CopyFrame: PROCEDURE [
    frame: FrameHandle,
    doc: DocInterchangeDefs.Doc,
    brush: FrameBrush,
    leftMargin, rightMargin, topMargin, bottomMargin: CARDINAL ← 0]
    RETURNS [
        ok: BOOLEAN,
        copy: FrameHandle];
```

frame is the frame to be copied. The document into which the frame is to be copied is identified by **doc**, and may be the same as, or different from, the document containing **frame**. As with **StartIllustrator**, **brush** determines the appearance of the frame border, and **leftMargin**, **rightMargin**, **topMargin**, and **bottomMargin** specify the frame margins.

76.2.1.2 Creating forms

After creating the frame by calling **StartIllustrator**, the client will typically add forms to it by calling various **Create*** procedures, described below. To each of these procedures the client passes the **FrameHandle** returned by **StartIllustrator** and the default properties to be used, as well as form-dependent parameters. (See Section 76.2.3.3 for details of default properties.) Each returns a **FormHandle**, used in subsequent operations on the form.

```
FormHandle: TYPE [2];

nilFormHandle: FormHandle = LOOPHOLE[LONG[NIL]];
```

The procedures used to create forms are divided into several groups: basic forms, clusters, trajectories and shapes, and text frames. These are described in separate sections below, followed by a section on copying an existing form.

Basic forms

```
CreatePoint: PROCEDURE [
    frame: FrameHandle,
    locn: Locn,
    defaultProps: DefaultProps]
    RETURNS [FormHandle];
```

CreatePoint creates a point form at the location specified by **locn**.

```
CreateLine: PROCEDURE [
    frame: FrameHandle,
    locnA, locnB: Locn,
    defaultProps: DefaultProps]
    RETURNS [FormHandle];
```

CreateLine creates a line form with start and end points specified by **locnA** and **locnB**.

```
CreateRectangle: PROCEDURE [
    frame: FrameHandle,
    pointA, pointC: Locn,
    defaultProps: DefaultProps]
    RETURNS [FormHandle];
```

CreateRectangle creates a rectangle form whose size and placement are determined by specifying two diagonally opposed vertices of the rectangle, **pointA** and **pointC**. The order in which these vertices are specified is not significant.

```
CreateParallelogram: PROCEDURE [
    frame: FrameHandle,
    pointA, pointB, pointC: Locn,
    defaultProps: DefaultProps]
    RETURNS [FormHandle];
```

CreateParallelogram creates a parallelogram form whose size, shape and placement are determined by **pointA**, **pointB** and **pointC**, these being any three consecutive vertices of the parallelogram.

```
CreateEllipse: PROCEDURE [
    frame: FrameHandle,
    pointA, pointB, pointC: Locn,
    defaultProps: DefaultProps]
    RETURNS [FormHandle];
```

CreateEllipse creates an ellipse form whose size, shape and placement are determined by **pointA**, **pointB** and **pointC**, these being any three consecutive vertices of the bounding parallelogram for the ellipse.

Clusters

A cluster is a form which is itself a grouping of forms, created by joining the forms of which it is to be composed. The resultant cluster can then be manipulated as a single entity. Note that clusters may themselves be part of another cluster.

```
Join: PROCEDURE [enumerateProc: EnumerateFormsProc, count: CARDINAL ← 0]
    RETURNS [FormHandle];
```

```
EnumerateFormsProc: TYPE = PROCEDURE [formCallBack: AcceptFormProc];
AcceptFormProc: TYPE = PROCEDURE [form: FormHandle];
```

The **enumerateProc**, supplied to **Join** by the client, is called back to obtain the forms which are to comprise the cluster. For each of these forms, the client should call the **formCallBack**, as passed into the **enumerateProc**, to add it to the cluster.

count is the number of forms which will comprise the cluster. If **count** is zero, the **enumerateProc** will be called once to count the forms, and then called again to build the cluster itself.

Trajectories and shapes

A shape is a form which corresponds to a (possibly filled) geometric shape, and is composed of a number of trajectories. Each trajectory is composed of a sequence of edges, and may be either open or closed. A closed trajectory is analogous to a polygon, and an open trajectory to a polyline.

A shape is created by calling **CreateShape**.

```
CreateShape: PROCEDURE [
    frame: FrameHandle,
    defaultProps: DefaultProps]
    RETURNS [FormHandle];
```

frame is the frame within which the shape is to be created, and **defaultProps** contains the default properties for the form.

Trajectories are added to a shape by calling **AddTrajectory**.

```
AddTrajectory: PROCEDURE [trajectory, shape: FormHandle];
```

A trajectory is itself created by calling **CreateTrajectory**.

```
CreateTrajectory: PROCEDURE [
    frame: FrameHandle,
    edges: CARDINAL,
    closed: BOOLEAN ← TRUE,
    boundaryType: BoundaryType ← including,
    defaultProps: DefaultProps]
    RETURNS [currentForm: FormHandle];
```

frame is the frame within which the trajectory is to be created. **edges** specifies the number of edges in the trajectory, and **closed** specifies whether the trajectory is closed or open.

Each trajectory has a boundary type which may be either *including* or *excluding*. A boundary type of *including* will cause a shade or texture fill to fill inside the trajectory, whilst one of *excluding* will cause the fill to fill outside. This enables objects with "holes" to be constructed. Figure 76.1 illustrates a shape composed of a rectangular trajectory with boundary type *including* and a triangular trajectory with boundary type *excluding*, and filled with 50% gray.

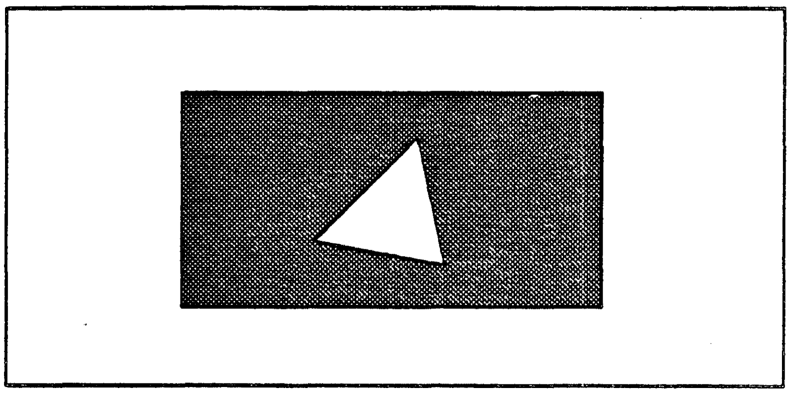


Figure 76.1 A shape with a "hole"

```
BoundaryType: TYPE = {including, excluding};
```

Edges are added to the trajectory by calling **AddEdge**.

```
AddEdge: PROCEDURE [trajectory: FormHandle, edge: Edge];
```

This causes **edge** to be added to **trajectory**. The edges of a trajectory may be lines, conics, bezier curves or arcs.

```
EdgeType: TYPE = {lineTo, conicTo, bezierTo, arcTo};

Edge: TYPE = LONG POINTER TO READONLY EdgeObject;
EdgeObject: TYPE = RECORD [
    smoothJoint: BOOLEAN ← FALSE,
    data: SELECT type: EdgeType FROM
        lineTo => [vertex: Locn],
        conicTo => [apex: Locn, eccentricity: Scalar, vertex: Locn],
        bezierTo => [apexA, apexB, vertex: Locn],
        arcTo => [direction: DirectionType ← clockwise,
            pointA, pointB, pointC, vertex: Locn],
        ENDCASE];
```

Since each edge of a trajectory starts at the end point (the **vertex**) of the previous edge, no starting point for an edge needs to be specified. For a closed trajectory, the first edge starts at the vertex of the last edge. For an open trajectory, the first edge defines the starting point of the trajectory (via its vertex), but is otherwise ignored.

A conic is specified by its apex and eccentricity, as illustrated in Figure 76.2. For

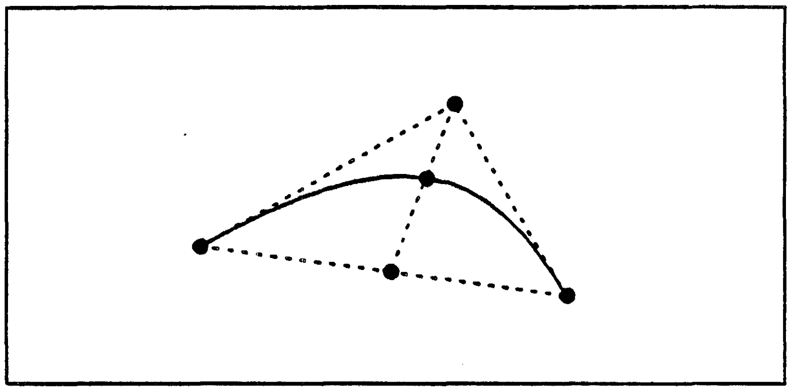


Figure 76.2 Specification of a conic

convenience, the following constant defines the eccentricity of a circle.

eccCircle: REAL = .414213; -- Sqrt [2.0] - 1.0

A bezier curve is specified by two bend points, **apexA** and **apexB**, as illustrated in Figure 76.3.

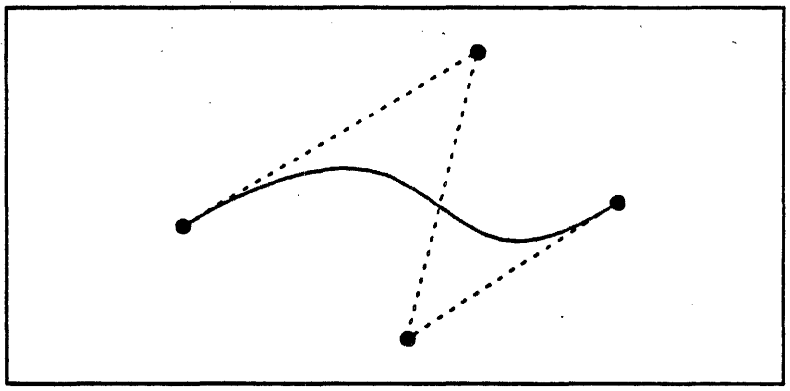


Figure 76.3 Specification of a bezier curve

An arc, which may be either circular or elliptical, is specified by its direction, which may be either clockwise or counter-clockwise, and three points on its bounding parallelogram, as illustrated in Figure 76.4.

DirectionType: TYPE = {clockwise, counterClockwise};

Note: In the current version, a trajectory can appear only inside a shape, only one trajectory per shape is permitted, and **boundaryType** is always taken as **including**.

Text frames

Text frames within a Pro Illustrator frame may be created in one of four different ways, depending upon the source of the text content itself. In each case, the client passes in

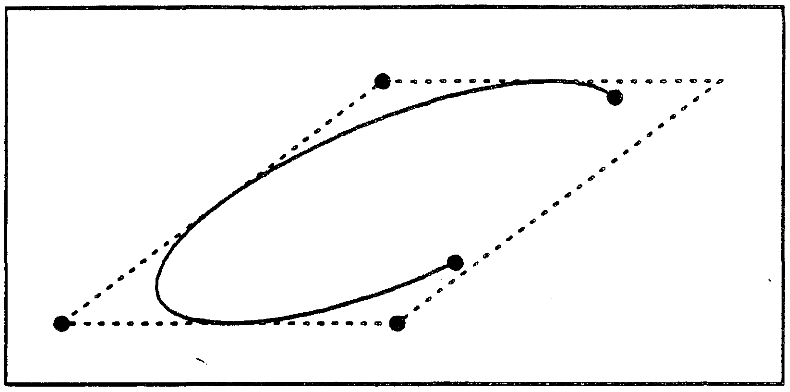


Figure 76.4 Specification of an arc

frame, which is the Pro Illustrator frame within which the text frame is to be created, **pointA** and **pointC**, which are two diagonally opposed vertices of the text frame, and **defaultProps**, which are the default properties to be used in creating the text frame. (See Section 76.2.3.3 for details of default properties.) The text frame form is returned in **FormHandle**.

Although the size and placement of a text frame is initially specified via **pointA** and **pointC**, this may be altered during creation if the frame has any varying sides , as determined from **defaultProps**. Since changing the frame to have fixed sides after it is created will not restore the frame's original size, the correct properties should be set within **defaultProps** before calling **CreateText***. (See **Property Groups** in Section 76.2.3.2 for details of text properties.)

```
CreateText: PROCEDURE [
    frame: FrameHandle,
    pointA, pointC: Locn,
    defaultProps: DefaultProps,
    string: XString.Reader ← NIL,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [FormHandle];
```

CreateText creates a text frame containing the text supplied in **string**, with paragraph and font properties determined by **paraProps** and **fontProps** respectively.

```
CreateTextFromTextType: PROCEDURE [
    frame: FrameHandle,
    pointA, pointC: Locn,
    defaultProps: DefaultProps,
    string: XString.Reader ← NIL,
    textContent: TextInterchangeDefs.Text ← NIL]
    RETURNS [FormHandle];
```

CreateTextFromTextType creates a text frame containing the text supplied in **textContent** preceded by that supplied in **string**, if any. This allows, for example, the copying of text from another text frame.

```
CreateTextFromField: PROCEDURE [
    frame: FrameHandle,
    pointA, pointC: Locn,
    defaultProps: DefaultProps,
    string: XString.Reader ← NIL,
    fieldContent: DocInterchangeDefs.Field ← NIL,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [FormHandle];
```

CreateTextFromField creates a text frame containing the text supplied in **fieldContent**, preceded by that supplied in **string**, if any, with paragraph and font properties determined by **paraProps** and **fontProps** respectively. This allows the copying of text from a field within a ViewPoint document into a text frame within a Pro Illustrator frame.

```
CreateTextFromCaption: PROCEDURE [
    frame: FrameHandle,
    pointA, pointC: Locn,
    defaultProps: DefaultProps,
    string: XString.Reader ← NIL,
    captionContent: DocInterchangeDefs.Caption ← NIL,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [FormHandle];
```

CreateTextFromCaption creates a text frame containing the text supplied in **captionContent**, preceded by that supplied in **string**, if any, with paragraph and font properties determined by **paraProps** and **fontProps** respectively. This allows the copying of text from a frame caption into a text frame within a Pro Illustrator frame.

Copying a form

A form may also be created by copying an existing form.

```
CopyForm: PROCEDURE [
    form: FormHandle,
    toFrame: FrameHandle ← nilFrameHandle]
    RETURNS [FormHandle];

nilFrameHandle: FrameHandle = LOOPHOLE[LONG[NIL]];
```

CopyForm creates and returns a copy of **form** within frame **toFrame**. If **toFrame** is **nilFrameHandle**, the form will be copied within the same frame

76.2.1.3 Posting forms

For a form to be visible within a displayed or printed Pro Illustrator frame, it must be posted after its creation. However, forms within a cluster should not themselves be posted, since posting the cluster achieves the correct result.

```
PostFormAtTop: PROCEDURE [form: FormHandle];
```

PostFormAtTop causes form to be imaged on top of all forms previously created within this frame.

A form may be unposted, or removed from the list of forms to be imaged within the frame, by calling **UnPostForm**.

```
UnPostForm: PROCEDURE [form: FormHandle];
```

76.2.1.4 Form transformations

After its creation, a variety of transformations may be applied to a form. Each of the following procedures performs a transformation on the form passed in, returning a boolean value indicating whether the form was actually changed or whether in fact the identity transformation took place.

```
Magnify: PROCEDURE [
    form: FormHandle,
    pin: Locn,
    scale: Scalar]
    RETURNS [changed: BOOLEAN];
```

Magnify causes **form** to be magnified by a factor of **scale**, centered around the pinned point, **pin**. Everything, including line widths, line endings and point widths, is magnified in proportion.

Note: While text frames can be magnified (subject to automatic constraints if they have varying edges), the text inside them is not. Also, the length of segments in dashed lines is not affected by magnification.

```
Mirror: PROCEDURE [
    form: FormHandle,
    pin: Locn,
    axis: Axis]
    RETURNS [changed: BOOLEAN];
```

```
Axis: TYPE = {x, y, origin, yEqualsX, yEqualsMinusX};
```

Mirror causes **form** to be reflected in the axis specified by **axis** and passing through the pinned point, **pin**. If **axis** is **origin**, then the reflection occurs through the pinned point, **pin**.

Note: Mirroring in the **yEqualsX** and **yEqualsMinusX** axes are currently not supported.

```
Rotate: PROCEDURE [
    form: FormHandle,
    pin: Locn,
    sin, cos: Scalar]
    RETURNS [changed: BOOLEAN];
```

Rotate causes **form** to be rotated by Θ degrees around the pinned point, **pin**, where **sin** and **cos** are the sine and cosine, respectively, of Θ.

Note: Embedded frames (currently only text frames) are not rotated, but instead keep their orientation and are moved along the path of the rotation.

```
Scale: PROCEDURE [
    form: FormHandle,
    pin: Locn,
    xScaleFactor: Scalar,
    yScaleFactor: Scalar]
    RETURNS [changed: BOOLEAN];
```

Scale causes **form** to be scaled by a factor of **xScaleFactor** in the X axis, and **yScaleFactor** in the Y axis, centered around the pinned point, **pin**. Line widths and line endings are not affected.

Note: While text frames can be scaled (subject to automatic constraints if they have varying edges), the text inside them is not.

```
Shear: PROCEDURE [
    form: FormHandle,
    pin: Locn,
    shearFactor: Scalar,
    scaleFactor: Scalar,
    axis: Axis]
    RETURNS [changed: BOOLEAN];
```

Shear causes **form** to be sheared by a factor of **shearFactor** in the axis **axis**, and scaled by a factor of **scaleFactor** in the orthogonal axis, centered around the pinned point, **pin**.

Note: Shearing in the **yEqualsX** and **yEqualsMinusX** axes are currently not supported. Embedded frames (currently only text frames) are not sheared, but instead keep their orientation and are moved along the path of the shear.

```
Translate: PROCEDURE [
    form: FormHandle,
    xOffset: Scalar,
    yOffset: Scalar,
    guidePtLocn: Locn ← origin,
    mode: TranslateMode ← relative]
    RETURNS [changed: BOOLEAN];

TranslateMode: TYPE = {absolute, relative};
```

If **mode** is **absolute**, **Translate** causes **form** to be translated by an amount such as to position the guide point, **guidePtLocn**, at the point (**xOffset**, **yOffset**). If **mode** is **relative**, **form** is translated by **xOffset** in the X axis and **yOffset** in the Y axis.

76.2.1.5 Finish routines

When all the desired forms have been added to the frame, the client should call **FinishIllustrator**, passing in the **FrameHandle** originally returned by **StartIllustrator**.

```
FinishIllustrator: PROCEDURE [frame: FrameHandle]
    RETURNS [illustrator: DocInterchangeDefs.Instance];
```

The client will typically pass **illustrator** to **DocInterchangeDefs.AppendAnchoredFrame** to insert the frame into a document.

76.2.2 Reading graphics

This section details the procedures used to perform basic enumeration of the forms within a Pro Illustrator frame. Interrogation of the properties of forms is achieved through use of the procedures described in the next section.

The client enumerates the forms within a Pro Illustrator frame by calling **Enumerate**.

```
Enumerate: PROCEDURE [frame: FrameHandle, proc: FormProc]
    RETURNS [stopped: BOOLEAN ← FALSE];

FormProc: TYPE = PROCEDURE [form: FormHandle] RETURNS [stop: BOOLEAN ← FALSE];
```

frame is the handle of the Pro Illustrator frame to be enumerated. The client-supplied **proc** is called back for each form in the frame, in bottom to top order. If **proc** returns **stop** = TRUE at any time, the enumeration is terminated, and **Enumerate** returns to the client with **stopped** = TRUE.

In order to enumerate the forms within a cluster, or trajectories within a shape, the client should call the following procedure.

```
EnumerateSubForms: PROCEDURE [form: FormHandle, proc: AcceptFormProc];
```

The client-supplied **proc** is called back for each form in the cluster, or trajectory within the shape.

In order to enumerate the edges within a trajectory, the client should call the following procedure.

```
EnumerateEdges: PROCEDURE [form: FormHandle, proc: EdgeProc]
    RETURNS [stopped: BOOLEAN ← FALSE];

EdgeProc: TYPE = PROCEDURE [edge: Edge] RETURNS [stop: BOOLEAN ← FALSE];
```

The client-supplied **proc** is called back for each edge in the trajectory. If **proc** returns **stop** = TRUE at any time, the enumeration is terminated, and **EnumerateEdges** returns to the client with **stopped** = TRUE.

76.2.3 Properties

Previous sections of this chapter have dealt with the creation and enumeration of frames and forms in their basic state. This section details the properties applicable to frames and to each type of form, and describes the procedures available to set and interrogate those properties.

76.2.3.1 Frame properties

The following procedures allow the client to get and set various properties of a Pro Illustrator frame. The **Get*** procedures take a **FrameHandle** as a parameter, and return the appropriate type of information as their result. The **Set*** procedures take a **FrameHandle** and the appropriate information as parameters.

```
GetFrameHandle: PROCEDURE [anchoredFrame: DocInterchangeDefs.Instance]
    RETURNS [FrameHandle];
```

GetFrameHandle returns the handle for an anchored Pro Illustrator frame within a document. **anchoredFrame** will typically have been passed to the client as a result of a call to **DocInterchangeDefs.Enumerate.**

```
GetBoundary: PROCEDURE [frame: FrameHandle] RETURNS [lowerLeft, upperRight: Locn];
```

```
SetBoundary: PROCEDURE [frame: FrameHandle, lowerLeft, upperRight: Locn];
```

GetBoundary and **SetBoundary** allow the client to get and set, respectively, the bounding box for a frame. The values **lowerLeft** and **upperRight** define the size of the frame and the coordinate system to be used within it.

```
GetFrameUnit: PROCEDURE [frame: FrameHandle] RETURNS [FrameUnit];
```

```
SetFrameUnit: PROCEDURE [frame: FrameHandle, frameUnit: FrameUnit];
```

GetFrameUnit and **SetFrameUnit** allow the client to get and set, respectively, the unit of measurement to be used in subsequent operations within the frame.

```
GetGridding: PROCEDURE [frame: FrameHandle] RETURNS [GriddingInfo];
```

```
SetGridding: PROCEDURE [frame: FrameHandle, gridding: GriddingInfo];
```

GetGridding and **SetGridding** allow the client to get and set, respectively, the parameters affecting the use of a grid within the frame.

```
GriddingInfo: TYPE = LONG POINTER TO READONLY GriddingObject;
GriddingObject: TYPE = RECORD [
    xGrid, yGrid: Scalar ← 0.0,
    rGrid, thetaGrid: Scalar ← 0.0,
    cartesianGridOrigin: Locn ← origin,
    polarGridOrigin: Locn ← origin,
    thetaGridOffset: Scalar ← 0.0,
    displayGrid: BOOLEAN ← FALSE,
    gridStyle: GridStyle ← none,
    gridType: GridType ← none,
    xGridOn, yGridOn: BOOLEAN ← FALSE,
    rGridOn, thetaGridOn: BOOLEAN ← FALSE];

GridStyle: TYPE = {tic, dot, none};
GridType: TYPE = {linear, angular, none};

origin: Locn = [0.0, 0.0];
```

Both cartesian and polar grids are provided for within a Pro Illustrator frame, that which is active (if any) being indicated by **gridType**. Whether or not the grid is visible to the user is determined by **displayGrid**, and the style of grid points, when visible, is specified by **gridStyle**.

Note: **gridStyle = tic** is not supported in the current version.

For a cartesian grid, X and Y grid control is independently specifiable. Whether or not X or Y positioning is to be constrained by the grid is specified by **xGridOn** and **yGridOn** respectively. The grid spacing is specified by **xGrid** and **yGrid**, measured from the grid origin, **cartesianGridOrigin**.

For a polar grid, r and Θ grid control is independently specifiable. Whether or not r or Θ positioning is to be constrained by the grid is specified by **rGridOn** and **thetaGridOn** respectively. The grid spacing is specified by **rGrid** measured from the grid origin, **polarGridOrigin**, and by **thetaGrid** measured in degrees around the grid origin, starting at **thetaGridOffset** degrees.

```
GetTopForm: PROCEDURE [frame: FrameHandle] RETURNS [FormHandle];

GetBottomForm: PROCEDURE [frame: FrameHandle] RETURNS [FormHandle];
```

GetTopForm and **GetBottomForm** return the handle of the top and bottom form, respectively, within the imaging order of the frame whose handle is passed in.

76.2.3.2 Form properties

The following procedures allow the client to get and set various properties of a form. These procedures are divided into three groups, reflecting their applicability to all types of forms, to specific types of forms, or to specific types of properties. A further section is devoted to the subject of line dash styles.

Generic form properties

These procedures return to the client information about the form in its context within the frame. They are applicable to all type of forms.

```
GetParentFrame: PROCEDURE [form: FormHandle] RETURNS [FrameHandle];
```

GetParentFrame returns the handle of the Pro Illustrator frame within which the form exists.

GetFormType: PROCEDURE [form: FormHandle] RETURNS [FormType];

FormType: TYPE = {
    point, line, rectangle, parallelogram, ellipse,trajectory, shape, cluster, text, frame};

GetFormType returns the type of the form whose handle is passed in.

GetFormAbove: PROCEDURE [form: FormHandle, circular: BOOLEAN ← FALSE]
    RETURNS [FormHandle];

GetFormBelow: PROCEDURE [form: FormHandle, circular: BOOLEAN ← FALSE]
    RETURNS [FormHandle];

GetFormAbove and **GetFormBelow** return the form above and below, respectively, the form whose handle is passed in, within the imaging order of the frame. Ordinarily, **GetFormAbove** returns NIL if **form** is the top form,and **GetFormBelow** returns NIL if **form** is the bottom form. However, if **circular** = TRUE, both functions "wrap around", that is, **GetFormAbove** of the top form returns the bottom form, and **GetFormBelow** of the bottom form returns the top form.

Specific form properties

Each of these procedures returns to the client the pertinent information for the particular type of form whose handle is passed in.

GetPointData: PROCEDURE [form: FormHandle] RETURNS [point: Locn];

GetPointData returns the location of the point passed in.

GetLineData: PROCEDURE [form: FormHandle] RETURNS [pointA, pointB: Locn];

GetLineData returns the start and end points of the line passed in.

GetParallelogramData: PROCEDURE [form: FormHandle]
    RETURNS [isRectangle: BOOLEAN, pointA, pointB, pointC: Locn];

GetParallelogramData returns three consecutive vertices of the parallelogram passed in, and indicates whether or not the parallelogram is in fact a rectangle.

GetEllipseData: PROCEDURE [form: FormHandle] RETURNS [pointA, pointB, pointC: Locn];

GetEllipseData returns three consecutive vertices of the bounding parallelogram for the ellipse passed in.

GetClusterData: PROCEDURE [form: FormHandle] RETURNS [count: CARDINAL];

GetClusterData returns, for the shape passed in, a count of the number of forms of which it is composed.

GetShapeData: PROCEDURE [form: FormHandle] RETURNS [count: CARDINAL];

GetShapeData returns, for the shape passed in, a count of the number of trajectories of which it is composed.

```
GetTrajectoryData: PROCEDURE [form: FormHandle]
    RETURNS [count: CARDINAL, closed: BOOLEAN, boundaryType: BoundaryType];
```

GetTrajectoryData returns the number of edges in the trajectory, an indication of whether it is closed or not, and the boundary type, for the trajectory passed in.

```
GetTextData: PROCEDURE [doc: DocInterchangeDefs.Doc, form: FormHandle]
    RETURNS [text: TextInterchangeDefs.Text];
```

GetTextData returns, for the text frame passed in, the text content for that frame.

Property groups

There are several groups of properties which are not directly connected to a particular type of form, but which may in fact be associated with several different types of forms. For each such group, three procedures are provided, viz **Has*Props**, **Get*Props** and **Set*Props**. For a given form, these procedures allow the client to determine whether the form possesses the properties, to get the properties, and to set them, respectively.

Has*Props procedures return TRUE if the form has that particular type of property – that is, if the property is appropriate for the form – and FALSE otherwise. For example, line properties are appropriate for lines, but area properties are not. The client should call the appropriate **Has*Props** procedure to determine the appropriateness of properties before calling **Get*Props** or **Set*Props**.

Get*Props procedures return the form property if that property is appropriate and consistent. For example, lines have line properties, but a cluster of lines might have inconsistent (i.e. different) line properties. If the property is inappropriate, NIL is returned; if it is inconsistent or does not exist, the value **undefined*Props** is returned.

Note that both line end properties and area properties are appropriate for shape forms, but that line end properties are used only on open trajectories, and area properties only on closed trajectories.

Area properties are normally applicable to rectangles, parallelograms, ellipses, closed trajectories and text frames, and may be applicable to clusters.

```
HasAreaProps: PROCEDURE [form: FormHandle] RETURNS [BOOLEAN];

GetAreaProps: PROCEDURE [form: FormHandle] RETURNS [AreaPropsObject];

SetAreaProps: PROCEDURE [form: FormHandle, areaProps: AreaProps];

AreaProps: TYPE = LONG POINTER TO AreaPropsObject;
AreaPropsObject: TYPE = RECORD [
    areaTexture: AreaTexture,
    areaColor: Color,
    transparent: Boolean];
```

```
Boolean: TYPE = {true, false, undefined};
```

An area has a texture and a color, and may be either transparent or opaque. A variety of basic textures are provided, and these may be used singly or in combination to achieve the desired effect.

```
AreaTexture: TYPE = RECORD [
    -- line --
    lineVert, lineHoriz, lineNW, lineNE,
    -- dash --
    dashNW, dashNE, brick, crossweave,
    -- other --
    stipple, whiteArc, wave, squareDot: Boolean];
```

As a convenience, noAreaTexture allows the client to specify the absence of any texture within an area.

```
noAreaTexture: AreaTexture = [
    false, false, false, false, false, false, false, false, false, false, false, false];
```

Color, as a shade of gray, is specified as a value in the range 0 to 1, where 0 and 1 represent white and black respectively.

```
Color: TYPE = Scalar;
```

The following definition represents no specification of area properties. Note that this is different from specification of no area properties.

```
undefinedAreaProps: AreaPropsObject = [
    undefinedAreaTexture, undefinedScalar, undefined];

undefinedAreaTexture: AreaTexture = [
    undefined, undefined, undefined, undefined, undefined, undefined,
    undefined, undefined, undefined, undefined, undefined, undefined];

undefinedScalar: Scalar = Real.LargestNumber;
```

Graphic properties are applicable to all form types.

```
HasGraphicProps: PROCEDURE [form: FormHandle] RETURNS [BOOLEAN];

GetGraphicProps: PROCEDURE [form: FormHandle] RETURNS [GraphicPropsObject];

SetGraphicProps: PROCEDURE [form: FormHandle, graphicProps: GraphicProps];

GraphicProps: TYPE = LONG POINTER TO GraphicPropsObject;
GraphicPropsObject: TYPE = RECORD [
    fixedAngle: Boolean,
    fixedShape: Boolean,
    printable: Boolean];
```

printable determines whether or not the form should be printed in addition to being displayed on the screen.

Note: **fixedAngle** and **fixedShape** are not implemented in the current release.

The following definition represents no specification of graphic properties.

```
undefinedGraphicProps: GraphicPropsObject = [undefined, undefined, undefined];
```

Point properties are applicable to points, and may be applicable to clusters.

```
HasPointProps: PROCEDURE [form: FormHandle] RETURNS [BOOLEAN];

GetPointProps: PROCEDURE [form: FormHandle] RETURNS [PointPropsObject];

SetPointProps: PROCEDURE [form: FormHandle, pointProps: PointProps];

PointProps: TYPE = LONG POINTER TO PointPropsObject;
PointPropsObject: TYPE = RECORD [
    pointMarker: PointMarker,
    pointWidth: Scalar,
    pointColor: Color];
```

A point has a marker type, a width and a color. A variety of markers are provided. Width, which is the effective diameter of the point, is specified in the current frame units. As for areas, color is specified as a value in the range 0 to 1, where 0 and 1 represent white and black respectively.

```
PointMarker: TYPE = {
    -- solid --
    solidSquare, solidCircle, solidDiamond, solidUpTriangle,
    solidDownTriangle, solidRightTriangle, solidLeftTriangle,
    -- open --
    openSquare, openCircle, openDiamond, openUpTriangle,
    openDownTriangle, openRightTriangle, openLeftTriangle,
    -- nDot --
    squareNDot, circleNDot, diamondNDot,
    -- misc --
    solidDiscRising, solidDiscSetting, solidStar,
    x, plus, outlineDiscRising, outlineDiscSetting,
    openStar, starburst, check,
    undefined};
```

The following definition represents no specification of point properties.

```
undefinedPointProps: PointPropsObject = [
    undefined, undefinedScalar, undefinedScalar];
```

Line properties are applicable to lines, rectangles, parallelograms, ellipses, trajectories and text frames, and may be applicable to clusters.

```
HasLineProps: PROCEDURE [form: FormHandle] RETURNS [BOOLEAN];
```

```
GetLineProps: PROCEDURE [form: FormHandle] RETURNS [LinePropsObject];

SetLineProps: PROCEDURE [form: FormHandle, lineProps: LineProps];

LineProps: TYPE = LONG POINTER TO LinePropsObject;
LinePropsObject: TYPE = RECORD [
    lineTexture: LineTexture,
    dashStyle: DashStyle,
    lineWidth: Scalar,
    lineColor: Color];
```

A line has a texture, a width and a color. If the line texture is dashed, the line has a dash style. Dash styles are described in the next section.

```
LineTexture: TYPE = {solid, dashed, none, undefined};
```

The following definition represents no specification of line properties.

```
undefinedLineProps: LinePropsObject = [
    undefined, NIL, undefinedScalar, undefinedScalar];
```

Line end properties are applicable to lines and trajectories, and may be applicable to clusters.

```
HasLineEndProps: PROCEDURE [form: FormHandle] RETURNS [BOOLEAN];

GetLineEndProps: PROCEDURE [form: FormHandle] RETURNS [LineEndPropsObject];

SetLineEndProps: PROCEDURE [form: FormHandle, lineEndProps: LineEndProps];

LineEndProps: TYPE = LONG POINTER TO LineEndPropsObject;
LineEndPropsObject: TYPE = RECORD [
    endingA: LineEnding,
    endSizeA: Scalar,
    fixedOrVaryingA: FixedOrVarying,
    endingB: LineEnding,
    endSizeB: Scalar,
    fixedOrVaryingB: FixedOrVarying];
```

For each end of the line, the **LineEndPropsObject** specifies the type and size of the line ending, and whether it is fixed or variable. The size of a fixed line ending is specified by **size**, and remains unchanged when the line width itself is scaled. **size** represents the line width for which the arrowhead is proportionately correct. For a variable line ending, **size** is ignored, and the actual size is determined from the line width. Variable line endings are scaled along with the line itself.

```
LineEnding: TYPE = {
    -- basic --
    extended, flush, rounded,
    -- solid arrows --
    arrowSolidMiddleFlatBack, arrowSolidShortFlatBack,
    arrowSolidEquilFlatBack, arrowSolidSkinnyFlatBack,
    arrowSolidSmallVBack, arrowSolidLargeVBack,
    -- outline arrows --
    arrowOutlineMiddleFlatBack, arrowOutlineShortFlatBack,
    arrowOutlineEquilFlatBack, arrowOutlineSkinnyFlatBack,
    arrowOutlineSmallVBack, arrowOutlineLargeVBack,
    -- V arrows --
    arrowVLong, arrowVFlatTop,
    -- misc --
    solidCircle, outlineCircle, break, slash,
    -- double arrows
    doubleMiddleFlatBack, doubleEquilFlatBack, doubleSmallVBack,
    undefined};
```

```
FixedOrVarying: TYPE = {fixed, varying, undefined};
```

The following definition represents no specification of line end properties.

```
undefinedLineEndProps: LineEndPropsObject = [
    undefined, undefinedScalar, undefined, undefined, undefinedScalar, undefined];
```

Text properties are applicable to text frames, and may be applicable to clusters.

```
HasTextProps: PROCEDURE [form: FormHandle] RETURNS [BOOLEAN];
```

```
GetTextProps: PROCEDURE [form: FormHandle] RETURNS [TextPropsObject];
```

```
SetTextProps: PROCEDURE [form: FormHandle, textProps: TextProps];
```

```
TextProps: TYPE = LONG POINTER TO TextPropsObject;
TextPropsObject: TYPE = RECORD [
    leftEdge, rightEdge, topEdge, bottomEdge: FixedOrVarying,
    ctrlPtPlace: CtrlPtPlace,
    leftMargin, rightMargin, topMargin, bottomMargin: Scalar];
```

Each edge of a text frame may be either fixed or varying. When an edge is specified as varying, its position is determined by the textual content of the frame, whereas the position of a fixed edge will not be altered in this way. The margin for each edge is independently specificable, and the positioning of control points for the text frame is determined by ctrlPtPlace.

```
CtrlPtPlace: TYPE = {baseline, center, baselineAndCenter, undefined};
```

When ctrlPtPlace is center, control points for the text frame are positioned the same as for a rectangle. When ctrlPtPlace is baseline, the control points in the middle of the left and right sides and in the center of the frame are positioned along the baseline of the first line of text.

Note: baselineAndCenter is not supported in the current version.

The following definition represents no specification of text properties.

```
undefinedTextProps: TextPropsObject = [
    undefined, undefined, undefined, undefined, undefined,
    undefinedScalar, undefinedScalar, undefinedScalar, undefinedScalar];
```

Dash styles

Pro Illustrator allows the creation and use of arbitrary dash styles for lines and trajectories. Due to this arbitrary nature, a special mechanism exists for their manipulation, the details of which are described in this section.

A dash pattern can be considered as a sequence of pairs of arbitrary length, each pair comprising a dash and a gap. Hence a regular dashed line would consist of a single pair, with the dash and gap of equal length, and a dash-dot line would consist of two pairs, the two dash lengths being different.

The client creates a new dash style by calling **CreateDashStyle**.

```
CreateDashStyle: PROCEDURE [
    zone: UNCOUNTED ZONE,
    fit: DashFit,
    enumerateProc: EnumeratePatternProc]
    RETURNS [DashStyle];

DashStyle: TYPE = LONG POINTER;
```

The storage for the new dash style will be allocated from **zone**. **fit** determines the behaviour of the dash pattern within a line.

```
DashFit: TYPE = {fromOneEnd, centered, stretchToFit, stretchToVertices};
```

fromOneEnd causes the dash pattern to be applied, repeatedly if necessary, without stretching, and starting with the first dash at the first vertex of the trajectory. **centered** causes the dash pattern to be applied without stretching and starting at the center of the trajectory. **stretchToFit** causes the dash pattern to be stretched as necessary so that each end of the trajectory has a full dash. **stretchToVertices** causes the dash pattern to be stretched as necessary, separately for each segment, so that each vertex of the trajectory has a full dash.

Note: Only **fit** = **fromOneEnd** is currently implemented.

The client-supplied **enumerateProc** is called back to obtain the pairs comprising the dash pattern.

```
EnumeratePatternProc: TYPE = PROCEDURE [dashCallBack: AcceptDashProc];

AcceptDashProc: TYPE = PROCEDURE [dash, gap: Scalar, dashEnd: DashEnd ← flush];
```

The client should call **dashCallBack** for each dash-gap pair of the dash pattern. **dashEnd** determines the appearance of the end of the dash.

Note: Currently, the unit of measure for dash patterns is always **point** regardless of the units in effect for the frame.

```
DashEnd: TYPE = {flush, extended, round};
```

flush causes the dash to end at precisely the point at which its length indicates. **extended** causes the dash to be extended by half its width in the direction of the trajectory at that point. **round** causes the the dash to appear with a semicircular end, whose diameter equals the line width and whose center coincides with the dash end.

A dash style may also be created by copying an existing dash style.

```
CopyDashStyle: PROCEDURE [zone: UNCOUNTED ZONE, dashStyle: DashStyle]
    RETURNS [DashStyle];
```

CopyDashStyle returns a copy of **dashStyle** whose storage is allocated from **zone**.

When a dash style is no longer required it should be destroyed, and its storage freed, using the following procedure.

```
DestroyDashStyle: PROCEDURE [zone: UNCOUNTED ZONE, ptr: LONG POINTER TO DashStyle];
```

zone must be the same as that which was passed to **CreateDashStyle** or **CopyDashStyle** when the dash style was created.

In order to interrogate the construction of an existing dash style, the client should call **ReadDashStyle**.

```
ReadDashStyle: PROCEDURE [dashStyle: DashStyle, proc: AcceptDashProc ← NIL]
    RETURNS [DashFit];
```

The client-supplied **proc** is called back for each dash-gap pair in **dashStyle**, and the fit for the dash style is returned returned by **ReadDashStyle**.

The equality or otherwise of two dash styles is determined using the following procedure.

```
DashStylesEqual: PROCEDURE [style1, style2: DashStyle] RETURNS [BOOLEAN];
```

DashStylesEqual returns TRUE if all elements of **style1** and **style2** are equal, and FALSE otherwise. Note that this is not the same as comparing **style1** and **style2** directly, since the two patterns may be exactly equivalent without being one and the same.

76.2.3.3 Default properties

Default properties are used in the creation of forms within a frame, and comprise a global set of property settings not specific to any particular type of form. The client will typically construct a single set of default values, and use this each time a form is created by calling **Create***. This section describes the procedures whereby the client can create and subsequently manipulate default properties.

A default property record is created by calling the following procedure.

```
CreateDefaults: PROCEDURE [z: UNCOUNTED ZONE] RETURNS [defaultProps: DefaultProps];

DefaultProps: TYPE = LONG POINTER TO DefaultPropsObject;

DefaultPropsObject: TYPE;
```

CreateDefaults returns a pointer to a default property object whose value consists of the set of standard Pro Illustrator defaults for each group of properties. The storage for the object is allocated from the client-supplied zone, **z**.

When a default property record is no longer required it should be destroyed, and its storage freed, using the following procedure.

```
DestroyDefaults: PROCEDURE [defaultProps: LONG POINTER TO DefaultProps];
```

For each group of properties described under Specific Property Types in Section 76.2.3.2 above, the client can get or set the defaults within **defaultProps** using the **Get*** and **Set*** procedures listed below. Each **Get*** procedure returns a complete ***PropsObject**, and each **Set*** procedure takes a ***Props**, as appropriate.

```
GetDefaultAreaProps: PROCEDURE [defaultProps: DefaultProps]
   RETURNS [AreaPropsObject];

SetDefaultAreaProps: PROCEDURE [
   areaProps: AreaProps, defaultProps: DefaultProps];

GetDefaultGraphicProps: PROCEDURE [defaultProps: DefaultProps]
   RETURNS [GraphicPropsObject];

SetDefaultGraphicProps: PROCEDURE [
   graphicProps: GraphicProps, defaultProps: DefaultProps];

GetDefaultPointProps: PROCEDURE [defaultProps: DefaultProps]
   RETURNS [PointPropsObject];

SetDefaultPointProps: PROCEDURE [
   pointProps: PointProps, defaultProps: DefaultProps];

GetDefaultLineProps: PROCEDURE [defaultProps: DefaultProps]
   RETURNS [LinePropsObject];

SetDefaultLineProps: PROCEDURE [
   lineProps: LineProps, defaultProps: DefaultProps];

GetDefaultLineEndProps: PROCEDURE [defaultProps: DefaultProps]
   RETURNS [LineEndPropsObject];

SetDefaultLineEndProps: PROCEDURE [
   lineEndProps: LineEndProps, defaultProps: DefaultProps];

GetDefaultTextProps: PROCEDURE [defaultProps: DefaultProps]
   RETURNS [TextPropsObject];

SetDefaultTextProps: PROCEDURE [
   textProps: TextProps, defaultProps: DefaultProps];
```

After changing the frame units using **SetFrameUnit**, the client should update any default property records by calling the following procedure.

```
UpdateDefaultFrameUnits: PROCEDURE [
    defaultProps: DefaultProps, frameUnit: FrameUnit];
```

UpdateDefaultFrameUnits will cause the values in **defaultProps** to be represented in **frameUnit** units, so that they are correctly interpreted in subsequent **Create*** operations.

76.2.4 Errors

```
ErrorCode: TYPE = {formNotPosted, unknown};
```

```
Error: ERROR [code: ErrorCode];
```

Error is raised only by **UnPostForm**. When the form is already unposted, **Error** will be raised with **code = formNotPosted**. Any other error causes **Error** to be raised with **code = unknown**.

76.3 Index of Interface Items

77

TableInterchangeDefs

77.1 Overview

TableInterchangeDefs allows clients to read the contents of a table, create a new table, or add information to an existing table. This interface should be used in conjunction with **DocInterchangeDefs** or **GraphicsInterchangeDefs**.

A table is described by three sets of properties: table properties, column properties, and row properties. Table properties include the name of the table, a description of table headers and the number of columns and rows in the table; column properties include whether the columns are divided, and the alignment of text within the columns; and row properties include information about how the text is aligned within a given row. The actual content of a table is included with the row information.

Most records below have spare fields for future use. When specifying values for these, it is important to use zero if you do not know of a correct value to use.

All dimension values are measured in micas. Clients may use the ViewPoint interface **UnitConversion** to convert to and from mica values.

77.1.1 Table building

To create a new table, the client should start by calling **StartTable**. This procedure takes table properties and column properties as parameters, and returns a table handle. **Handle** points to **Object**, which is a record that contains, along with table-related data, a pointer to the actual table content (See section 77.2.8: Diagram of table structure). Initially, the row properties have default values and the table has no content; the client should initialize row properties and content after the call to **StartTable**.

To add content to the table, the client can pass the table handle to **AppendRow**, which adds new information to the table. When all of the rows have been added, the final step is to call **FinishTable**, which creates the final structure for the table. Once the table is created, the client can pass this table to the procedures in **DocInterchangeDefs** or **GraphicsInterchangeDefs** to add it to a document.

FinishTable returns a **DocInterchangeDefs.Instance** for the table, which the client can pass to **DocInterchangeDefs.AppendAnchoredFrame** or **GraphicsInterchangeExtra2Defs.AddTable**. This instance is the table frame's content; the rest of a frame's properties (like captions and border style) are handled by **DocInterchangeDefs** and **GraphicsInterchangeDefs**.

To add information to an existing table, the client should call **StartExistingTable** instead of **StartTable**. This procedure also returns a table handle, which the client can then pass to **AppendRow** and **FinishTable**. **StartExistingTable** takes an **DocInterchangeDefs.Instance** as a parameter; the client will typically call **TableSelectionDefs.TableFromSelection** to get the currently selected table as a value of type **DocInterchangeDefs.Instance**.

77.1.2 Table reading

To read the contents of a table, the client typically starts by calling **Enumerate**. **Enumerate** takes as arguments a table object (**DocInterchangeDefs.Instance**) and a record of three call back procedures: a **TableProc**, a **ColumnsProc**, and a **RowProc**.

Enumerate will call the **TableProc** and the **ColumnsProc** once for a given table; these procedures obtain the table and column properties. Since the content of the table is stored with the rows, **Enumerate** will call the **RowProc** once for each row in the table.

There is a also a procedure **EnumerateSpecificRows**, which is just like **Enumerate** except that it enumerates a specific list of rows within a table rather than the entire table. **EnumerateSpecificRows** will call the **RowProc** once for each row in the specified range of rows.

77.2 Interface Items

77.2.1 Table properties

A **TablePropsRec** describes the properties of a table and its headers.

```
TableProps: TYPE = LONG POINTER TO TablePropsRec;

TablePropsRec: TYPE = RECORD [
    name: XString.Reader,
    fillinByRow,
    fixedRows,
    fixedColumns: BOOL,
    numberOfColumns,
    numberOfRows: NATURAL,
    visibleHeader,
    repeatHeader,
    repeatTopCaption,
    repeatBottomCaption: BOOL,
    borderLine,
    dividerLine: Line,
    horizontalAlignment: HeaderAlignment,
    headerVerticalAlignment: VerticalAlignment,
    topHeaderMargin, bottomHeaderMargin: LONG CARDINAL,
    sortKeys: SortKeys,
    spare1: LONG CARDINAL];
    ascending: BOOL,
    spare1: LONG CARDINAL];
```

The **SortKeysRec** contains a sequence of optional **SortKeys** for a table or column. A column must be divided-repeating in order to have sort keys. Each **SortKey** contains the column's name, its **sortOrder** and whether to sort in ascending or descending order.

If **spare1** is not zero, it is assumed to be an **ExtraTableProps** pointer.

```
ExtraTableProps: TYPE = LONG POINTER TO ExtraTablePropsRec;

ExtraTablePropsRec: TYPE = RECORD [
    deferOnPaginate: BOOL,
    spare1: LONG CARDINAL];
```

deferOnPaginate indicates whether the pagination operation will defer the table frame to the next page if it will not fit on the current one. If **deferOnPaginate** is FALSE, the portion of the table that fits on the current page will be placed there, and the remainder will appear on successive pages.

spare1 in the **ExtraTablePropsRec** is for future use.

ExtraTableProps is currently defined in **TableInterchangeExtra1Defs**.

77.2.2 Column properties

```
ColumnInfo: TYPE = LONG POINTER TO ColumnInfoSeq;

ColumnInfoSeq: TYPE = RECORD [SEQUENCE length: CARDINAL OF ColumnInfoRec];

ColumnInfoRec: TYPE = RECORD [
    headerEntryRec: HeaderEntryRec,
    name, description: XString.Reader,
    divided: BOOL,
    subcolumns: NATURAL,
    repeating: BOOL,
    subcolumnInfo: ColumnInfo,
    alignment: HorizontalAlignment,
    tabOffset, -- Micas! (different from DocInterchangePropsDefs.TabStop)
    width,
    leftMargin,
    rightMargin: LONG CARDINAL,
    type: DocInterchangePropsDefs.FieldChoiceType,
    required: BOOL,
    language: MultiNational.Language,
    format: XString.Reader,
    stopOnSkip: BOOL,
    range: XString.Reader,
    length: CARDINAL,
    skipText: XString.Reader,
    skipChoice: DocInterchangePropsDefs.SkipIfChoiceType,
    fillin: XString.Reader,
```

name is the name of the table.

fillinByRow determines what happens when the user presses the NEXT key. If **fillInByRow** is TRUE, pressing the NEXT key advances through the table one row at a time, and the table is expanded by rows. In this case, the number of columns is fixed and the number of rows can be either fixed or varying. If **fillinByRow** is FALSE, then pressing the NEXT key advances through the table one column at a time, and the table is expanded by columns. In this case, the number of rows is fixed and the number of columns can be either fixed or varying. **fixedRows** and **fixedColumns** indicate whether the user can change the number of rows and columns in the table.

numberOfColumns and **numberOfRows** are used as hints for **StartTable**.

visibleHeader indicates whether there should be a visible header at the top of the table; **repeatHeader**, **repeatTopCaption**, **repeatBottomCaption** indicates whether or not to repeat these items on every page if the table occupies multiple pages.

borderLine describes the table border (not the frame border), and **dividerLine** describes the line between the header row and the rest of the table. A line can have a width anywhere from one pixel to six pixels.

```
Line: TYPE = RECORD [
    linestyle: Linestyle,
    linewidth: Linewidth];

Linestyle: TYPE = MACHINE DEPENDENT{
    none(0), solid, dashed, dotted, double, broken, firstAvailable,lastAvailable(255)};

Linewidth: TYPE = MACHINE DEPENDENT {w1(0), w2(1), w3(2), w4(3), w5(4), w6(5)};
```

horizontalAlignment and **headerVerticalAlignment** specify the alignment of the text within a header.

```
HeaderAlignment: TYPE = HorizontalAlignment [left..right];

HorizontalAlignment: TYPE = MACHINE DEPENDENT{left(0), center(1), right(2), decimal(3)};

VerticalAlignment: TYPE = MACHINE DEPENDENT {
    flushtop(0), centered(1), flushbottom(2)};
```

topHeaderMargin and **bottomHeaderMargin** specify the amount of white space that should appear between above and below each header element.

```
SortKeys: TYPE = LONG POINTER TO SortKeysRec;

SortKeysRec: TYPE = RECORD [
    length: CARDINAL,
    spare1: LONG CARDINAL,
    keys: SEQUENCE maxLength: CARDINAL OF SortKey];

SortKey: TYPE = RECORD [
    columnName: XString.Reader,
    sortOrder: XString.SortOrder,
     ascending: BOOL,
     spare1: LONG CARDINAL];
```

The **SortKeysRec** contains a sequence of optional **SortKeys** for a table or column. A column must be divided-repeating in order to have sort keys. Each **SortKey** contains the column's name, its **sortOrder** and whether to sort in ascending or descending order.

If **spare1** is not zero, it is assumed to be an **ExtraTableProps** pointer.

```
ExtraTableProps: TYPE = LONG POINTER TO ExtraTablePropsRec;

ExtraTablePropsRec: TYPE = RECORD [
     deferOnPaginate: BOOL,
     spare1: LONG CARDINAL];
```

deferOnPaginate indicates whether the pagination operation will defer the table frame to the next page if it will not fit on the current one. If **deferOnPaginate** is FALSE, the portion of the table that fits on the current page will be placed there, and the remainder will appear on successive pages.

spare1 in the **ExtraTablePropsRec** is for future use.

ExtraTableProps is currently defined in **TableInterchangeExtra1Defs**.

77.2.2 Column properties

```
ColumnInfo: TYPE = LONG POINTER TO ColumnInfoSeq;

ColumnInfoSeq: TYPE = RECORD [SEQUENCE length: CARDINAL OF ColumnInfoRec];

ColumnInfoRec: TYPE = RECORD [
     headerEntryRec: HeaderEntryRec,
     name, description: XString.Reader,
     divided: BOOL,
     subcolumns: NATURAL,
     repeating: BOOL,
     subcolumnInfo: ColumnInfo,
     alignment: HorizontalAlignment,
     tabOffset, -- Micas! (different from DocInterchangePropsDefs.TabStop)
     width,
     leftMargin,
     rightMargin: LONG CARDINAL,
     type: DocInterchangePropsDefs.FieldChoiceType,
     required: BOOL,
     language: MultiNational.Language,
     format: XString.Reader,
     stopOnSkip: BOOL,
     range: XString.Reader,
     length: CARDINAL,
     skipText: XString.Reader,
     skipChoice: DocInterchangePropsDefs.SkipIfChoiceType,
     fillin: XString.Reader,
    fillinRuns: DocInterchangePropsDefs.FontRuns,
    line: Line,
    sortKeys: SortKeys,
    spare1: LONG CARDINAL];
```

A **ColumnInfoSeq** describes all the columns of a table; a **ColumnInfoRec** describes one column in detail. Within a **ColumnInfoRec**, the most complicated field is a **headerEntryRec**; all of the other fields correspond directly to the fields on the property sheet that the user sees. The next section discusses the header properties, and section 77.2.4 discusses the remaining column properties.

For a more complete description of any of these properties, see the user documentation.

77.2.3 Column header properties

```
HeaderEntryRec: TYPE = RECORD [
    subHeaders: HeaderInfo,
    line: Line,
    singleLineHint: BOOL,
    spare1: LONG CARDINAL,
    content: EntryContent];
```

A **HeaderEntryRec** describes the textual content of a column header. Header text can contain any number of font and paragraph properties per column header.

subHeader describes the headers for each of the subcolumns. This field is only interesting if the column is **divided**. **subHeader** points to a sequence that contains a HeaderEntryRec for each subcolumn. Each subcolumn may in turn be subdivided, in which case that subcolumn's **HeaderEntryRec subHeader** field would point to another sequence.

```
HeaderInfo: TYPE = LONG POINTER TO HeaderInfoSeq;
```

```
HeaderInfoSeq: TYPE = RECORD [SEQUENCE length: CARDINAL OF HeaderEntryRec];
```

line describes the properties of line that divides the header from subheaders. **line** is only visible if the column is subdivided.

singleLineHint is a hint that the header only contains one line of text; this makes the code somewhat faster by simplifying the calculation of header size. If a client specifies a **singleLineHint** value of TRUE for a header entry and then appends more than one line of text, the resulting header entry will appear one line tall even though it has more text inside. The user can correct this by editing the text in that header entry, which causes the entry to recompute its height.

spare1 is for future use.

```
EntryContent: TYPE = RECORD [
    SELECT mode: * FROM
        read = > [obtainTextProc: ObtainTextProc, obtainTextData: ObtainTextData],
        write = > [fillInTextProc: FillInTextProc ← NIL, clientData: LONG POINTER ← NIL],
    ENDCASE];
```

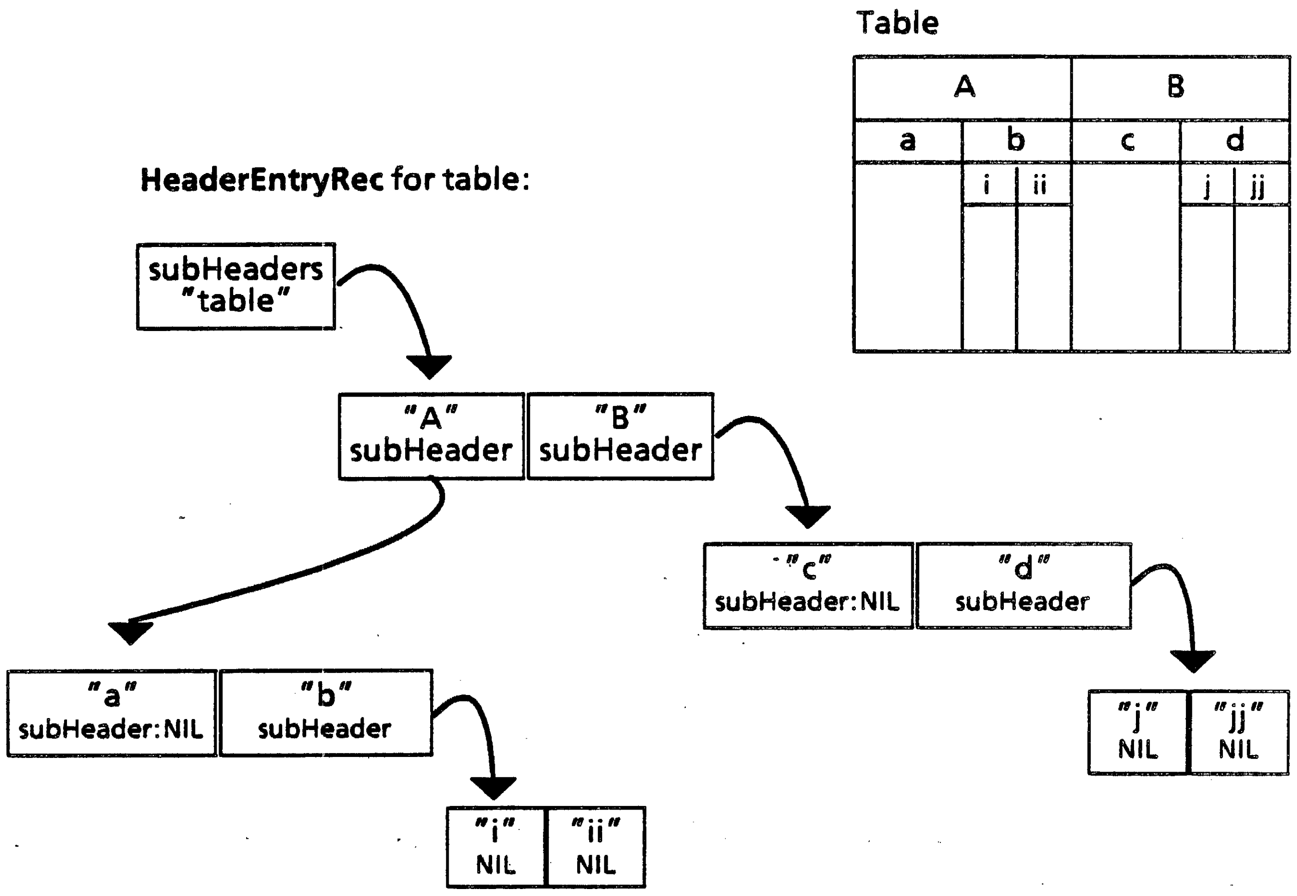


Figure 77.1 Table and HeaderEntryRec for table

```
ObtainTextData: TYPE[4];

ObtainTextProc: TYPE = PROC [obtainTextData: ObtainTextData]
    RETURNS [text: TextInterchangeDefs.Text];

FillInTextProc: TYPE = PROC [text: TextInterchangeDefs.Text, clientData: LONG POINTER];
```

content is a variant record that describes the content for a header entry. When enumerating a table, all the header entries will be of the form **[read[...]]**. The client may call the **ObtainTextProc** for an entry to obtain a **TextInterchangeDefs.Text** object, which may then be enumerated via **TextInterchangeDefs.EnumerateText**. If the client does call an **ObtainTextProc**, **TextInterchangeDefs.ReleaseText** must be called on the returned **TextInterchangeDefs.Text** object when the client is finished enumerating it.

When creating a table, the client must set all header entries to **[write[...]]**. The client may set the **fillInTextProc** to a proc to be called back to fill in the entry with text. **clientData** will be passed to the client's **fillInTextProc**. The client may default the **fillInTextProc** to **NIL** so that the entry will be empty.

77.2.4 Other column properties

name and **description** are the name and description of the table as it appears in the property sheet.

divided specifies whether the columns can be divided. **subcolumns** is the number of subcolumns; **repeating** indicates that subcolumns can have subrows, and **subcolumnInfo** is the recursive description of the subcolumns. **subcolumns**, **repeating**, and **subcolumnInfo** are ignored if **divided** is FALSE.

alignment describes the alignment of the text within a column.

tabOffset specifies where a decimal tab should be set, relative to the margin. **tabOffset** only applies if **alignment** = **decimal**. Like all other dimensions, **tabOffset** is in micas (this is different from a **DocInterchangePropsDefs.TabStopOffset**, which is measured in units of 1/72 inch).

width is the width of the column; **leftMargin** and **rightMargin** are the margins for the column. These values are also in micas.

type indicates the type of content that will appear in a column.

required indicates that the entry is required, and that the user must fill it in before proceeding to another entry in the table.

language affects the format of date and amount fields. It is used when items are added to the paragraph.

format allows the user to define a format to which the data in the column must conform.

stopOnSkip When the user presses SKIP, the skipping action should stop at the next entry in this column.

range is used to define a specific range of acceptable entries for the column. Once defined, any entry not within the defined range is not acceptable. See the user documentation for information on how ranges are defined.

length allows the user to define the maxiumum number of characters that will be accepted in the column entries.

skipText and **skipChoice** defines the conditions under which an area may be skipped when the user presses NEXT. See the user documentation for more detail.

fillin and **fillinRuns** describe the fill-in rule for the column.

line describes the properties of the vertical line to the right of the column.

sortKeys describes the optional sort keys for the column.

spare1 is for future use.

77.2.5 Row content

```
RowContent: TYPE = LONG POINTER TO RowContentSeq;

RowContentSeq: TYPE = RECORD [
    topMargin, bottomMargin: LONG CARDINAL ← 0,
    line: Line ← [solid, w2],
    verticalAlignment: VerticalAlignment ← flushtop,
    spare1: LONG CARDINAL ← 0,
    rowdata: SEQUENCE length: CARDINAL OF RowEntryRec];
```

RowContentSeq describes row properties and content. The margins are the row margins; **line** is the properties of the line separating the rows. **verticalAlignment** specifies the alignment of text within a row. **spare1** is for future use. **rowdata** describes the content.

```
RowEntryRec: TYPE = RECORD [
    subRows: SubRows,
    singleLineHint: BOOL,
    spare1: LONG CARDINAL,
    content: EntryContent];
```

A **RowEntryRec** describes the textual content of a given row entry.

```
SubRows: TYPE = LONG POINTER TO SubRowsRec;

SubRowsRec: TYPE = RECORD [
    length: CARDINAL,
    spare1: LONG CARDINAL ← 0,
    rows: SEQUENCE maxLength: CARDINAL OF RowContent];
```

SubRowsRec describes subrow properties and content. If **subRows** is non-NIL, then the rest of the **RowEntryRec** record is unused, since the information will be in the individual subrow records.

Note that subrows may only exist if the parent column is divided.

The remaining fields are as described for header properties.

77.2.6 Table building operations

77.2.6.1 Creating a new table

```
StartTable: PROC [
    doc: DocInterchangeDefs.Doc,
    props: TableProps,
    c: ColumnInfo]
    RETURNS [h: Handle];
```

StartTable creates a document table in doc. props describes the properties of the table itself; c describes the properties of the columns. The Handle that is returned contains a description of row properties and table content.

StartTable will raise DocInterchangeDefs.Error[documentFull] if the table and header row will not fit in the document. If StartTable raises this error, the table cannot be added to the document due to lack of storage space for structures.

StartTable returns a handle:

```
Handle: TYPE = LONG POINTER TO Object;

Object: TYPE = RECORD [
    zone: UNCOUNTED ZONE,
    table: DocInterchangeDefs.Instance,
    tableHeight: LONG CARDINAL,
    tableWidth: LONG CARDINAL,
    rc: RowContent,
    spare1: LONG CARDINAL,
    private: ARRAY [0..0) OF WORD];
```

zone is the zone from which dynamic storage specific to this operation is allocated. table is the table itself.

tableHeight is initially equal to the height of the header row and is updated after each call to AppendRow. tableHeight and tableWidth are in micas. rc points to a record used as temporary storage for a new row.

77.2.6.2 Opening an existing table

```
StartExistingTable: PROC [
    table: DocInterchangeDefs.Instance,
    hi: HeaderInfo ← NIL,
    rowPropsSource: NATURAL ← 0,
    deleteExistingRows: BOOL ← TRUE,
    numberOfRowsHint: NATURAL ← 0]
    RETURNS [h: Handle];
```

StartExistingTable sets things up to append rows to an existing table. table is the table object. The table passed in to StartExistingTable is often obtained from a call to TableSelectionDefs.TableFromSelection, which returns the current selection as a table.

hi describes the desired properties for the table headers. If hi = NIL then the existing column headers are used.

rowPropsSource is the index of a row in the table; this is the row from which the default properties are taken. The rows are numbered from [0..*n*]. The horizontal alignment for each entry is taken from first new paragraph character in the corresponding element of the first row.

deleteExistingRows indicates whether the implementation should delete the existing contents of the table before adding new information. **numberOfRowsHint** is a hint about the number of rows that the table will contain; this is for efficiency purposes.

Like **StartTable**, **StartExistingTable** returns a **Handle**, which the client can then pass to **AppendRow**.

This procedure will raise **DocInterchangeDefs.Error[readonlyDoc]** if the document is read-only.

77.2.6.3 Appending rows

```
AppendRow: PROC [h: Handle, rc: RowContent];
```

AppendRow adds the row described by **rc** to the table described by **h**. Typically, **h** will be a handle obtained from either **StartTable** or **StartExistingTable**.

RowContent is as described in section 77.2.5.

77.2.6.4 Finishing a table

```
FinishTable: PROC [h: Handle]
    RETURNS [
    table: DocInterchangeDefs.Instance,
    tableWidth, tableHeight: LONG CARDINAL];
```

The client should call **FinishTable** when it is through editing a table. The **table** that is returned is intended to be passed as the **content** argument to **DocInterchangeDefs.AppendAnchoredFrame**, or as the **table** parameter to **GraphicsInterchangeExtra2Defs.AddTable**. This operation deletes **h.zone**. **tableWidth** and **tableHeight** are in micas.

77.2.6.5 Miscellaneous utilities

```
MaxTableElements: PROC RETURNS [NATURAL];
```

This procedure returns an estimate of the number of table cells that could reside in a document that has no other types of structures within. Clients may use this value to estimate how big a table could be created in a document.

```
DefaultFontProps: PROC [font: DocInterchangePropsDefs.FontProps];
```

```
DefaultParaProps: PROC [para: DocInterchangePropsDefs.ParaProps];
```

These procedures take a properties record and fill in reasonable default values. These operations are similar to the ones defined in **DocInterchangePropsDefs**.

```
GetTablePropsFromName: PROC [
    doc: DocInterchangeDefs.Doc,
    tableName: XString.Reader,
     tableProps: TableProps,
     zone: UNCOUNTED ZONE];
```

doc is the document from which to retrieve the properties of the table specified by **tableName**.

tableProps.name will be NIL.but the remainder of **tableProps** will contain the table's properties.

If **tableProps.sortKeys** is not NIL, it will be allocated from **zone**.

77.2.7 Table reading operations

```
EnumerateTable: PROC [
     table: DocInterchangeDefs.Instance,
     procs: EnumProcs,
     clientData: LONG POINTER ← NIL];

EnumProcs: TYPE = LONG POINTER TO EnumProcsRec;

EnumProcsRec: TYPE = RECORD [
     tableProc: TableProc ← NIL,
     columnsProc: ColumnsProc ← NIL,
     rowProc: RowProc ← NIL];
```

To parse the contents of a table, clients call **EnumerateTable** or **EnumerateSpecificRows**. **EnumerateTable** takes as parameters a table handle and a record of callback procedures: one for table properties, one for column properties, and one for row properties.

```
TableProc: TYPE = PROC [
     clientData: LONG POINTER,
     props: TableProps]
     RETURNS [stop: BOOL ← FALSE];

ColumnsProc: TYPE = PROC [
     clientData: LONG POINTER,
     columns: ColumnInfo]
     RETURNS [stop: BOOL ← FALSE];

RowProc: TYPE = PROC [
     clientData: LONG POINTER,
     content: RowContent]
     RETURNS [stop: BOOL ← FALSE];
```

Enumerate calls the **TableProc** and the **ColumnsProc** once, passing in the appropriate property information. Because the content of the table is stored with the rows, **EnumerateTable** calls the **rowProc** once for each row in the table.

Each of these callback procedures has a boolean return value. If **stop** is ever returned TRUE, then the enumeration will stop.

```
EnumerateSpecificRows: PROC [
    tr: TableRows,
    procs: EnumProcs,
    clientData: LONG POINTER ← NIL];

TableRows: TYPE = RECORD [
    table, firstRow, lastRow: DocInterchangeDefs.Instance];
```

EnumerateSpecificRows describes a certain subset of rows in a table. As with **EnumerateTable**, the **tableProc** and the **columnsProc** will each be called once to describe the appropriate properties; the column information will describe the columns intersecting the described rows. The **RowProc** will be called once for each row in [**firstRow..lastRow**].

77.2.8 Diagram of table structure

Figure 77.2 is a diagram of a **table** structure. **RowContent** is a pointer to **RowContentSeq**. **table** is a record that contains two pointers to the actual instance of the table. (Note that **table** itself is not a pointer.)

77.2.9 Constants

The following constants can be used to initialize the various table properties to reasonable default values.

```
nullExtraTableProps: ExtraTablePropsRec = [
    deferOnPaginate: TRUE,
    spare1: 0];
```

nullExtraTableProps is currently defined in **TableInterchangeExtra1Defs**.

```
nullLine: Line = [linestyle: solid, linewidth: w1];

nullSortKey: SortKey = [
    columnName: NIL,
    sortOrder: standard,
    ascending: TRUE,
    spare1: 0];

nullColumnInfo: ColumnInfoRec = [
    headerEntryRec: nullHeaderEntry,
    name: NIL,
    description: NIL,
    divided: FALSE,
    subcolumns: 0,
    repeating: FALSE,
    subcolumnInfo: NIL,
    alignment: center,
    tabOffset: 0,
    width: 2540,
    leftMargin: 0,
    rightMargin: 0,
    type: any,
```

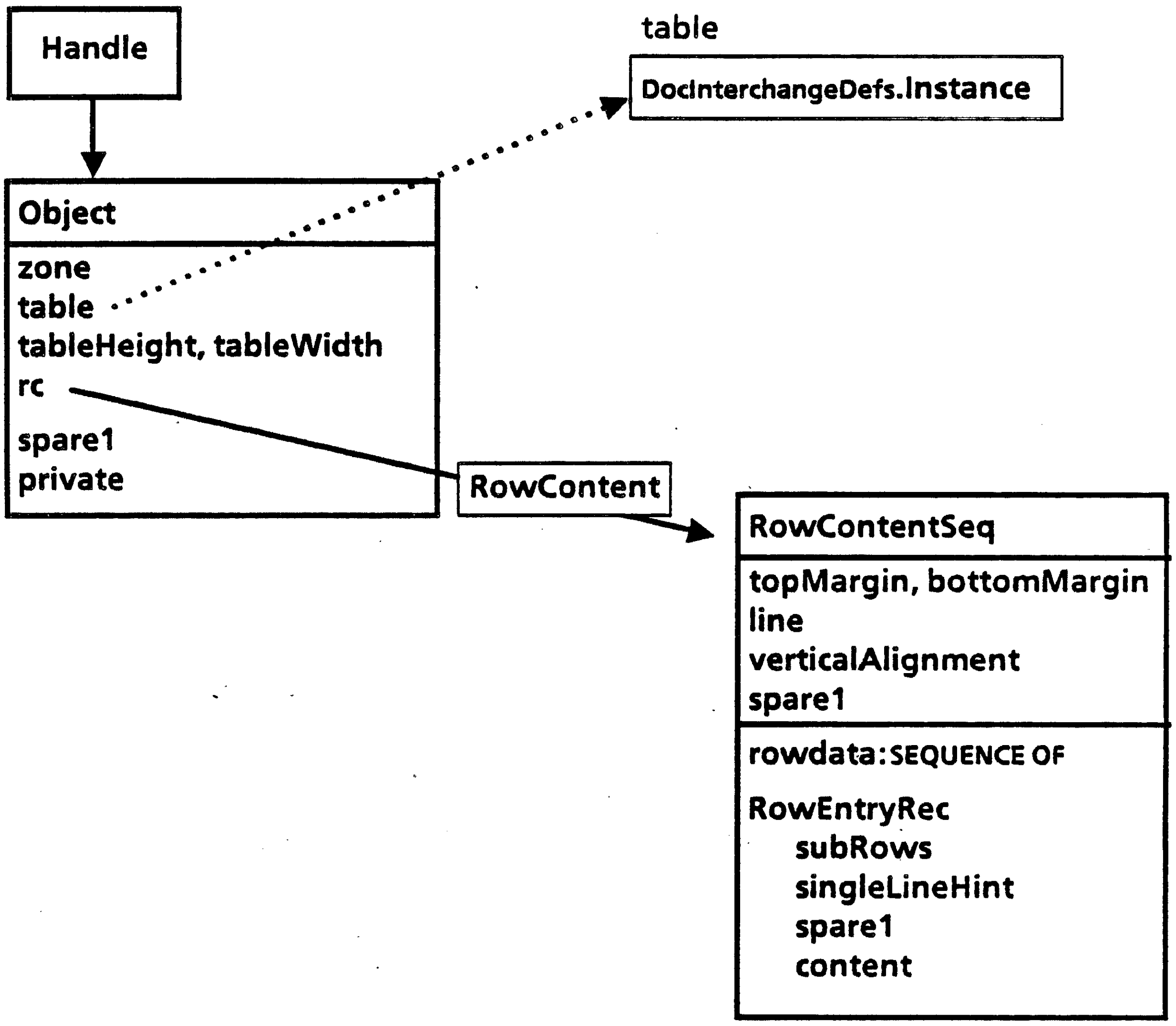


Figure 77.2 Diagram of Table Structure

```
    required: FALSE,
    language: USEnglish,
    format: NIL,
    stopOnSkip: FALSE,
    range: NIL,
      length: 0,
    skipText: NIL,
    skipChoice: empty,
    fillin: NIL,
    fillinRuns: NIL,
    line: [solid, w2],
    sortKeys: NIL,
    spare1: 0];

nullHeaderEntry: HeaderEntryRec = [
    subHeaders: NIL,
    line: [solid, w2],
    singleLineHint: FALSE,
    spare1: 0,
    content: [write[]]];

nullRowEntry: RowEntryRec = [
    subRows: NIL,
    singleLineHint: FALSE,
    spare1: 0,
    content: [write[]]];

nullTableProps: TablePropsRec = [
    name: NIL,
    fillinByRow: TRUE,
    fixedRows: FALSE,
    fixedColumns: TRUE,
    numberOfColumns: 0,
    numberOfRows: 0,
    visibleHeader: TRUE,
    repeatHeader: TRUE,
    repeatTopCaption: TRUE,
    repeatBottomCaption: TRUE,
    borderLine: [none, w1],
    dividerLine: [solid, w4],
    horizontalAlignment: center,
    headerVerticalAlignment: centered,
    topHeaderMargin: 0,
    bottomHeaderMargin: 0,
    sortKeys: NIL,
    spare1: 0];

tableRowsNil: TableRows = [
    DocInterchangeDefs.instanceNil,
    DocInterchangeDefs.instanceNil,
    DocInterchangeDefs.instanceNil];
```

tableRowsNil specifies a null value for **TableRows**. This value may not be passed into **EnumerateSpecificRows**. It may be returned from one of the operations in **TableSelectionDefs** under certain error conditions.

77.2.10 Errors

```
TableError: SIGNAL [type: ErrorType];

ErrorType: TYPE = MACHINE DEPENDENT{
    tableTooWide, tableTooTall, tableHeaderTooTall, firstAvailable, lastAvailable(255)};
```

tableTooWide	**StartTable** will raise this error if the specified table is too wide to fit in the document.
tableTooTall	**AppendRow** will raise this error if the specified table is too tall to fit in the document.

tableHeaderTooTall	**StartTable** will raise this error if the specified headers are too tall.

Do not call any Interchange operations from within a catch phrase of **TableError**.

77.3 Usage/Examples

Here is an example of a simple program that runs from the Attention Menu. It registers two commands: Make Table, which creates a new document with a table, and Add To Table, which adds four new rows to the selected table.

```
DIRECTORY
   ...;

TableExample: PROGRAM
   IMPORTSTableInterchangeDefs, TableSelectionDefs, TextInterchangeDefs, ... = {

   tableWidth: CARDINAL = 1600; --micas
   headerMargin: CARDINAL = 35 * 9; --micas; margin should be 9 pixels
   rowMargin: CARDINAL = 100;

   << Menu Proc for Create Table command. Creates new document, creates new table,
      appends table to document, and then adds document to desktop. >>
   MakeDocument: MenuData.MenuProc = {
      rows, columns: CARDINAL ← 3; --arbitrary
      doc: DocInterchangeDefs.Doc ← DocInterchangeDefs.StartCreation[
         paginateOption: none].doc;
      table: DocInterchangeDefs.Instance = BuildSimpleTable[doc, rows, columns];
      props: DocInterchangePropsDefs.FramePropsRecord ←
         DocInterchangePropsDefs.nullFrameProps;
      props.frameDims ← [tableWidth, tableWidth];
      [] ← DocInterchangeDefs.AppendAnchoredFrame[
         to: doc,
         type: table,
         anchoredFrameProps: @props,
         content: table];
      AddFileToDeskTop[doc];
      }; --MakeDocument

   << Create table inside doc with specified number of rows and columns.
      The content will be the string "abc." >>
   BuildSimpleTable: PROC [doc: DocInterchangeDefs.Doc, rows, columns: CARDINAL]
      RETURNS [table: DocInterchangeDefs.Instance ← DocInterchangeDefs.instanceNil] = {
      h: TableInterchangeDefs.Handle;
      contentString: XString.ReaderBody ← XString.FromSTRING["abc"L];
      c: TableInterchangeDefs.ColumnInfo ← Heap.systemZone.NEW[
         TableInterchangeDefs.ColumnInfoSeq[columns]];
      props: TableInterchangeDefs.TablePropsRec ← [
         name: NIL,
         fillinByRow: TRUE,
         fixedRows: FALSE,
         fixedColumns: TRUE,
         numberOfColumns: columns,
         numberOfRows: rows,
         visibleHeader: TRUE,
      repeatHeader: TRUE,
      repeatTopCaption: TRUE,
      repeatBottomCaption: TRUE,
      borderLine: [none, w1],
      dividerLine: [solid, w4],
      horizontalAlignment: center,
      headerVerticalAlignment: centered,
      topHeaderMargin: headerMargin,
      bottomHeaderMargin: headerMargin,
      sortKeys: NIL,
      spare1: 0];
   FOR i: CARDINAL IN [0..columns) DO
      c[i] ← TableInterchangeDefs.nullColumnInfo;
      c[i].width ← tableWidth;
      ENDLOOP;
   -- start creating table
   h ← TableInterchangeDefs.StartTable[doc: doc, props: @props, c: c];
   Heap.systemZone.FREE[@c];
   -- set row props and content
   h.rc.topMargin ← rowMargin;
   h.rc.bottomMargin ← rowMargin;
   FOR i: CARDINAL IN [0..rows) DO
      FOR j: CARDINAL IN [0..columns) DO
         h.rc[j] ← [
            subRows: NIL,
            singleLineHint: FALSE,
            spare1: 0,
            content: [write[fillInTextProc: FillInText, clientData: @contentString]]];
      ENDLOOP;
      TableInterchangeDefs.AppendRow[h, h.rc];
      ENDLOOP;
   RETURN [TableInterchangeDefs.FinishTable[h].table];
   }; -- BuildSimpleTable

<< Call-back procedure that writes text into the Text field of the table.
   The text to write is specified by the clientData argument. >>
FillInText: TableInterchangeDefs.FillInTextProc = {
   << PROC [text: TextInterchangeDefs.Text, clientData: LONG POINTER]; >>
   r: XString.Reader ← NARROW[clientData, XString.Reader];
   TextInterchangeDefs.AppendTextToText[
      to: text,
      text: r,
      textEndContext: XString.unknownContext];
   }; -- FillInText

AddFileToDeskTop: PROC [doc: DocInterchangeDefs.Doc] = {
   docFile: NSFile.Handle ← DocInterchangeDefs.FinishCreation[@doc].docFile;
   refDoc: NSFile.Reference = NSFile.GetReference[docFile];
   refDt: NSFile.Reference = StarDesktop.GetCurrentDesktopFile[];
   fileDt: NSFile.Handle = NSFile.OpenByReference[refDt];
   NSFile.Move[docFile, fileDt];
   NSFile.Close[fileDt];
   NSFile.Close[docFile];
   StarDesktop.AddReferenceToDesktop[refDoc]
   }; -- AddFileToDeskTop
```

```
<< Menu Proc for Add To Table command.
   Just adds four new blank rows to the selected table. >>
AddToTable: MenuData.MenuProc = {
   h: TableInterchangeDefs.Handle ← NIL;
   table: DocInterchangeDefs.Instance =
      TableSelectionDefs.TableFromSelection[];
   -- if current selection is not a table, then return. Otherwise,
   --add new rows to it. If doc is not editable, then return.
   IF table = DocInterchangeDefs.instanceNil THEN RETURN
   ELSE {
      h ← TableInterchangeDefs.StartExistingTable[
         table: table, deleteExistingRows: FALSE -- catch error if doc is not editable
         ! TableInterchangeDefs.TableError => GOTO Exit];
      THROUGH [0..4) DO
         TableInterchangeDefs.AppendRow[h, h.rc];
         ENDLOOP;
      };
   [] ← TableInterchangeDefs.FinishTable[h];
   EXITS Exit => NULL;
   }; -- AddToTable

Init: PROC = {
   makeTable: XString.ReaderBody ← XString.FromSTRING["MakeTable"L];
   addToTable: XString.ReaderBody ← XString.FromSTRING["AddToTable"L];
   Attention.AddMenuItem[
      MenuData.CreateItem[
         zone: Heap.systemZone,
         name: @makeTable,
         proc: MakeDocument]];
   Attention.AddMenuItem[
      MenuData.CreateItem[
         zone: Heap.systemZone,
         name: @addToTable,
         proc: AddToTable]];
   }; -- Init

Init[];
}.
```

77.4 Index of Interface Items

78

TableSelectionDefs

78.1 Overview

TableSelectionDefs provides procedures to obtain the current selection as a table, or a selection of rows within a table. This interface is meant to be used in conjunction with **TableInterchangeDefs**.

78.2 Interface Items

TableFromSelection: PROC RETURNS [DocInterchangeDefs.Instance];

TableFromSelection returns the current selection as an object of type **DocInterchangeDefs.Instance**. The client will typically pass this value to **TableInterchangeDefs.StartExistingTable**. If the current selection is not a table, **TableFromSelection** will return **DocInterchangeDefs.instanceNil**.

TableRowsFromSelection: PROC RETURNS [tr: TableInterchangeDefs.TableRows];

TableRowsFromSelection returns the current selection as a series of rows in a table. The client will typically pass this value as a parameter to **TableInterchangeDefs.EnumerateSpecificRows**. If the current selection is not one or more table rows, **TableRowsFromSelection** will return **TableInterchangeDefs.tableRowsNil**.

tableTarget: Selection.Target;

tableRowsTarget: Selection.Target;

```
TableFromValue: PROC [v: Selection.Value]
  RETURNS [DocInterchangeDefs.Instance];
```

```
TableRowsFromValue: PROC [v: Selection.Value]
  RETURNS [tr: TableInterchangeDefs.TableRows];
```

tableTarget, **tableRowsTarget**, **TableFromValue** and **TableRowsFromValue** are not currently implemented.

```
GetHostDocAccess: PROC [instance: DocInterchangeDefs.Instance]
    RETURNS [Access];

Access: TYPE = MACHINE DEPENDENT {readOnly(0), readWrite, (255)};
```

Access specifies whether or not the document is in edit mode.

78.3 Index of Interface Items

79

TextInterchangeDefs

79.1 Overview

TextInterchangeDefs provides procedures to create and enumerate text that resides in locations that **DocInterchangeDefs** does not define. This interface is used by **GraphicsInterchangeDefs** to handle the content of nested text frames and by **TableInterchangeDefs** to handle the content of header and row entries. **TextInterchangeDefs** also provides procedures to handle the content of anchored text frames.

All dimensions are in micas. Most records below have spare fields for future use. When specifying values for these, it is important to use zero if you do not know of a correct value to use.

79.2 Interface Items

79.2.1 Data types

The basic data structure of **TextInterchangeDefs** is **Text**, which is a pointer to an opaque text-containing object.

```
Text: TYPE = LONG POINTER TO TextObject;
```

```
TextObject: TYPE;
```

TFrameProps specify the properties of anchored text frames. Appending and enumerating text frames is covered later in this chapter.

```
TFrameProps: TYPE = LONG POINTER TO TFramePropsRec;
```

```
ReadonlyTFrameProps: TYPE = LONG POINTER TO READONLY TFramePropsRec;
```

```
TFramePropsRec: TYPE = RECORD [
   innerMargin: LONG CARDINAL,
   name, description: XString.Reader,
   spare1: LONG CARDINAL];
```

innerMargin is the uniform spacing between the text and the frame border. The client can vary the **innerMargin** within [0..264] micas ([0..7] pixels).

name and **description** are the name and description of the text frame as they appear in the property sheet.

The **spare1** field can be a pointer to a **TFrameExtraPropsRec** (see below). These extra properties are active only for text frames, not form fields. So on enumeration, the **spare1** field for form fields will be 0; on creation, form field **spare1** is ignored. When a text frame is encountered during enumeration, **spare1** will always be non-0, and may be LOOPHOLEd into a **ReadonlyTFrameExtraProps**. During creation, the implementation assumes that if a spare1 field is non-0, it should be treated as a **ReadonlyTFrameExtraProps** pointer. If it is 0, the implementation will use default values for the extra properties. These defaults come from the **nullTFrameExtraProps** constant.

```
TFrameExtraProps: TYPE = LONG POINTER TO TFrameExtraPropsRec;

ReadonlyTFrameExtraProps: TYPE = LONG POINTER TO READONLY TFrameExtraPropsRec;

TFrameExtraPropsRec: TYPE = RECORD [
    orientation: Orientation,
    lastLineJustify: BOOLEAN,
    autoHyphenate: BOOLEAN,
    spare0: PACKED ARRAY [3..15] OF CARDINAL [0..1],
    spare1: LONG CARDINAL];

Orientation: TYPE = {horizontal, vertical};
```

orientation indicates whether the text in the frame flows horizontally or vertically. Only Japanese text flows vertically.

lastLineJustify is only used in linked text frames. It specifies whether the last line of text in the frame is justified.

autoHyphenate is only used in linked text frames. It specifies whether the text in the frame is auto-hyphenated.

spare0 and **spare1** are for future use. They should be set to 0.

All of these extra properties are currently defined in **TextInterchangeExtra1Defs**.

```
nullTFrameExtraProps: TFrameExtraPropsRec = [
    orientation: horizontal,
    lastLineJustify: FALSE,
    autoHyphenate: FALSE,
    spare0: ALL [0],
    spare1: 0];

nullTFrameProps: TFramePropsRec = [
    innerMargin: 141, -- 4 pixels
    name: NIL,
    description: NIL,
    spare1: 0];
```

nullTFrameProps provides default initialization values for the TFramePropsRec.

79.2.2 Creating an Anchored Text Frame

StartTextInAnchoredFrame is used to begin appending text to the body of an anchored text frame. After an anchored text frame has been appended to a document via DocInterchangeDefs.AppendAnchoredFrame, StartTextInAnchoredFrame may be called to permit text to be appended to its body.

```
StartTextInAnchoredFrame: PROC [
    doc:DocInterchangeDefs.Doc,
    anchoredFrame: DocInterchangeDefs.Instance,
    props: ReadonlyTFrameProps]
    RETURNS [text: Text];
```

doc is the document containing the new anchored text frame.

anchoredFrame is the DocInterchangeDefs.Instance returned by the call to DocInterchangeDefs.AppendAnchoredFrame.

79.2.3 Append Operations

The following append* procedures are similar to the ones found in DocInterchangeDefs; the only difference is that these procedures append to Text objects.

```
AppendCharToText: PROC [
    to: Text,
    char: XChar.Character,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];
```

AppendChar appends one or more copies of the text character char to the specified Text object. nToAppend specifies the number of copies of the character that are to be appended; fontProps specifies the character properties.

```
AppendFieldToText: PROC [
    to: Text,
    fieldProps: DocInterchangePropsDefs.ReadonlyFieldProps,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [field:DocInterchangeDefs.Field];
```

AppendField appends a field to the specified Text object. AppendFieldToText returns a field; the client can then add information to the field by using the Field as a DocInterchangeDefs.TextContainer in other calls to DocInterchangeDefs.Append* routines. When the client is through with the field, it must release it via DocInterchangeDefs.ReleaseField.

Note that the client cannot set the fill-in order of the fields when they are appended to the document. This may be done via DocInterchangeDefs.AppendItemToFillInOrder.

```
AppendNewParagraphToText: PROC [
    to: Text,
    paraProps: DocInterchangePropsDefs.ReadonlyParaProps ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL,
    nToAppend: CARDINAL ← 1];
```

AppendNewParagraphToText appends one or more newParagraph characters to a Text object. nToAppend specifies the number of characters to be appended. paraProps and fontProps specify the properties for the paragraph. If paraProps is NIL, the new paragraph inherits the properties of the previous paragraph; however if this is the first new paragraph in the Text object, then it will have default properties.

Note that Text objects always contain at least one newParagraph character. The client does not have to provides these initial newParagraph characters; the TextInterchange implementation supplies them as required, although the client is free to append them anyway. The implementation ensures that if the client appends one at the start of the Text object, two won't appear. The client's paragraph and font properties on the newParagraph they appended will have precedence.

```
AppendTextToText: PROC [
    to: Text,
    text: XString.Reader,
    textEndContext: XString.Context,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL];
```

AppendTextToText appends the text with the specified properties to the Text object. For efficiency, the client should pass the appropriate textEndContext if it is known (just like XString.AppendReader). text may not contain newParagraph characters ([set: 0, code: 35B]). Use AppendNewParagraphToText to append these.

```
AppendTileToText: PROC [
    to: Text,
    type: Atom.ATOM,
    data: LONG POINTER ← NIL,
    fontProps: DocInterchangePropsDefs.ReadonlyFontProps ← NIL]
    RETURNS [tile: DocInterchangeDefs.Tile];
```

AppendTileToText is for future use. The tile type and data format are defined elsewhere, agreed upon by parties on either side of this interface.

79.2.4 Enumeration

To extract the content of a text object, clients call EnumerateText.

```
EnumerateText: PROC [
    text: Text,
    procs: TextEnumProcs,
    clientData: LONG POINTER ← NIL]
    RETURNS [dataSkipped: BOOLEAN];
```

procs is a pointer to a record containing client defined call-back procedures; these enumerate the various kinds of structures that can be found in **text**.

clientData is a client defined argument that will be passed to each of the call-back procedures.

```
TextEnumProcs: TYPE = LONG POINTER TO TextEnumProcsRecord;

TextEnumProcsRecord: TYPE = RECORD [
    fieldProc: DocInterchangePropsDefs.FieldProc ← NIL,
    newParagraphProc: DocInterchangePropsDefs.NewParagraphProc ← NIL,
    textProc: DocInterchangePropsDefs.TextProc ← NIL,
    tileProc: DocInterchangePropsDefs.TileProc ← NIL];
```

TextForAnchoredFrame is used when a client wants to enumerate the body of an anchored text frame.

```
TextForAnchoredFrame: PROC [
    doc: DocInterchangePropsDefs.Doc,
    content: DocInterchangePropsDefs.Instance,
    props: TFrameProps]
    RETURNS [text: Text];
```

doc is the document containing the anchored text frame to enumerate and **content** is the value passed to the **DocInterchangeDefs.AnchoredFrameProc**. The **Text** object that is returned may be passed to **EnumerateText**. After enumerating the text, the client must call **ReleaseText** on the text object returned by **TextForAnchoredFrame**.

TextForAnchoredFrame fills in **props** with text specific properties of the frame.

79.2.5 Releasing Text

The client should release the text object after calling **StartTextInAnchoredFrame** or **TextForAnchoredFrame**.

```
ReleaseText: PROC [textPtr: LONG POINTER TO Text];
```

79.2.6 Text Frame Link Order

This section discusses the data types and operations related to the text frame link order of a document. Every document may have an optional text frame fill-in order which defines the order in which text is poured into frames by pagination and by the "Fill Text Frames" command. This fill-in order is similar to the one for fields and tables.

All of the items declared in this section are currently defined in **TextInterchangeExtra1Defs**.

```
TextLinkItem: TYPE = LONG POINTER TO READONLY TextLinkItemRec;
```

```
TextLinkItemRec: TYPE = RECORD [
    name: XString.Reader,
    spare1: LONG CARDINAL];
```

Specifies an item in the linked text frame order.

```
TextLinkEnumProc: TYPE = PROC [item: TextLinkItem, clientData: LONG POINTER]
    RETURNS [stop: BOOL ← FALSE];
```

Callback proc for enumerating the text frame link order. See **EnumerateTextLink**.

```
EnumerateTextLink: PROC [
    doc: DocInterchangeDefs.Doc,
    proc: TextLinkEnumProc,
    clientData: LONG POINTER ← NIL];
```

Enumerates the text frames in the text frame link order.

```
ClearTextLink: PROC [doc: DocInterchangeDefs.Doc];
```

Clears the text frame link order of the document. May be called at any time. Usually called once just before a series of calls to **AppendItemToTextLink**.

```
AppendItemToTextLink: PROC [
    doc: DocInterchangeDefs.Doc,
    item: TextLinkItem];
```

Appends an item (a text frame) to the text frame link order. May be called at any time.

79.3 Example

The proper sequence of calls to append an anchored text frame having content is:

```
props: TextInterchangeDefs.TFramePropsRec ← [...];
[anchoredFrame, ...] ← DocInterchangeDefs.AppendAnchoredFrame[
   to: doc, type: text, ...];
text ← TextInterchangeDefs.StartTextInAnchoredFrame[doc, anchoredFrame, @props];
TextInterchangeDefs.AppendTextToText[text, reader, ...];
TextInterchangeDefs.ReleaseText[@text];
```

It is not mandatory that the client call **StartTextInAnchoredFrame** after appending an anchored frame. Failing to call **StartTextInAnchoredFrame** simply means that the anchored text frame will be empty, except for one new paragraph character that has default paragraph and font properties. Note that if the client does not call **StartTextInAnchoredFrame**, then the client should not call **ReleaseText**.

79.4 Index of Interface Items

Appendix A

System KeyNames and TIP Tables

A.1 Overview

This appendix contains generally useful information about how ViewPoint "sees" keys at the stimulus level and at the TIP level.

§A.2 lists KeyNames in terms known to level4 (dlion) and level5 (daybreak) stimlev, TIP names known to both the XDE and BWS world, and how all these relate to each other. The XDE comparison is designed to help the programmer who programs in both worlds and/or converts XDE tools to ViewPoint tools.

§A.3 deals with some TIP predicates that ViewPoint registers at boot time. These useful predicates may be used by clients as well as the BWS. For more information on TIP.Predicates see the TIP chapter Semantics of Tables.

The system **TIP** tables ViewPoint uses are listed to provide the programmer with a list of the productions available in the general-purpose tables. The Normal tables described in §A.4.1 are placed in the **TIPStar** watershed at boot time and are therefore available for use by application programs. (See the **TIPStar** interface for further information about the **TIPStar** watershed.) Clients are encouraged to use the productions in the Normal tables whenever possible rather than generating new tables. Examples of use are provided in §A.5.

A.2 KeyNames/TIP name mapping

LevelIVKeys.KeyName	LevelVKeys.KeyName (if different from LevelIV)	BWS TIP name	XDE TIP name * (if different from BWS)
notAKey(0)	null(0)	"null"	0
	Bullet(1)	"Bullet"	+
	SuperSub(2)	"SuperSub"	+
	Case(3)	"Case"	+
	Strikeout(4)	"Strikeout"	+
	KeypadTwo(5)	"KeypadTwo"	+
	KeypadThree(6)	"KeypadThree"	+
	SingleQuote(7)	"SingleQuote"	+
Keyset1(8)	KeypadAdd(8)	"KeypadAdd"	+
Keyset2(9)	KeypadSubtract(9)	"KeypadSubtract"	+
Keyset3(10)	KeypadMultiply(10)	"KeypadMultiply"	+
Keyset4(11)	KeypadDivide(11)	"KeypadDivide"	+
Keyset5(12)	KeypadClear(12)	"KeypadClear"	+
MouseLeft(13)	Point(13)	"Point"	
MouseRight(14)	Adjust(14)	"Adjust"	
MouseMiddle(15)	Menu(15)	"Menu"	
Five(16)		"Five"	
Four(17)		"Four"	
Six(18)		"Six"	
E(19)		"E"	
Seven(20)		"Seven"	
D(21)		"D"	
U(22)		"U"	
V(23)		"V"	
Zero(24)		"Zero"	
K(25)		"K"	
Minus(26)	Dash(26)	"Dash"	
P(27)		"P"	
Slash(28)		"Slash"	
Font(29)		"FONT"	"BackSlash"
Same(30)		"SAME"	"PASTE"
BS(31)		"BS"	
Three(32)		"Three"	
Two(33)		"Two"	
W(34)		"W"	
Q(35)		"Q"	
S(36)		"S"	
A(37)		"A"	
Nine(38)		"Nine"	
I(39)		"I"	
X(40)		"X"	
O(41)		"O"	
L(42)		"L"	
Comma(43)		"Comma"	
CloseQuote(44)	Quote(44)	"Quote"	#
RightBracket(45)		"RightBracket"	
Open(46)		"OPEN"	"STUFF"

LevelIVKeys.KeyName	LevelVKeys.KeyName (if different from LevelIV)	BWS TIP name	XDE TIP name * (if different from BWS)
Keyboard(47)	Special(47)	"SPECIAL"	"COMMAND"
One(48)		"One"	
Tab(49)		"TAB"	"COMPLETE" #
ParaTab(50)		"PARATAB"	"TAB"
F(51)		"F"	
Props(52)		"PROPS"	"CONTROL"
C(53)		"C"	
J(54)		"J"	
B(55)		"B"	
Z(56)		"Z"	
LeftShift(57)		"LeftShift"	
Period(58)		"Period"	
SemiColon(59)		"SemiColon"	
NewPara(60)		"NewPara"	"Return"
OpenQuote(61)		"OpenQuote"	"Arrow" #
Delete(62)		"DELETE"	
Next(63)		"NEXT"	
R(64)		"R"	
T(65)		"T"	
G(66)		"G"	
Y(67)		"Y"	
H(68)		"H"	
Eight(69)		"Eight"	
N(70)		"N"	
M(71)		"M"	
Lock(72)		"LOCK"	
Space(73)		"Space"	
LeftBracket(74)		"LeftBracket"	
Equal(75)		"Equal"	
RightShift(76)		"RightShift"	
Stop(77)		"STOP"	"USERABORT"
Move(78)		"MOVE"	
Undo(79)		"UNDO"	
Margins(80)		"MARGINS"	"DOIT"
R9(81)	KeypadSeven(81)	"KeypadSeven"	+
L10(82)	KeypadEight(82)	"KeypadEight"	+
L7(83)	KeypadNine(83)	"KeypadNine"	+
L4(84)	KeypadFour(84)	"KeypadFour"	+
L1(85)	KeypadFive(85)	"KeypadFive"	+
A9(86)	English(86)	"English"	J
R10(87)	KeypadSix(87)	"KeypadSix"	+
A8(88)	Katakana(88)	"Katakana"	J
Copy(89)		"COPY"	
Find(90)		"FIND"	
Again(91)		"AGAIN"	
Help(92)		"HELP"	#
Expand(93)		"EXPAND"	
R4(94)	KeypadOne(94)	"KeypadOne"	+

LevelIVKeys.KeyName	LevelVKeys.KeyName (if different from LevelIV)	BWS TIP name	XDE TIP name * (if different from BWS)
D2(95)	DiagnosticBitTwo(95)	"DiagnosticBitTwo"	0
D1(96)	DiagnosticBitOne(96)	"DiagnosticBitOne"	0
Center(97)		"CENTER"	"MENU"
T1(98)	KeypadZero(98)	"KeypadZero"	+
Bold(99)		"BOLD"	"SCROLLBAR"
Italics(100)	Italic(100)	"ITALICS"	"JFIRST"
Underline(101)		"UNDERLINE"	"JSELECT"
Superscript(102)		"SUPERSCRIPT"	"RESERVED" #
Subscript(103)		"SUBSCRIPT"	"CLIENT1" #
Smaller(104)		"SMALLER"	"CLIENT2"
T10(105)	KeypadPeriod(105)	"KeypadPeriod"	+
R3(106)	KeypadComma(106)	"KeypadComma"	+
Key47(107)	LeftShiftAlt(107)	"LeftShiftAlt"	J
A10(108)	DoubleQuote(108)	"DoubleQuote"	+
Defaults(109)		"DEFAULTS"	"ATTENTION" #
A11(110)	Hiragana(110)	"Hiragana"	J
A12(111)	RightShiftAlt(111)	"RightShiftAlt"	J

*Key: (0 not on any keyboard) (J JStar only) (+ daybreak only) (# dandelion only)

A.3 ViewPoint Registered TIP.Predicates

You will notice in the following tables the use of several TIP.Predicates. These predicates are registered by ViewPoint at boot time and are available for use by clients as well as the system. Most of these predicates are used to dertermine within a TIP.Table which physical keyboard is attached to the workstation. The following is the list of predicate.atoms and their meaning. See the Normal tables themselves for examples of use. For more information about TIP.Predicates see §47.3.3 PredicateIdent.

level4	matches any of the DLion keyboards (American, European or Japanese)
aLevel4	American DLion keyboard
eLevel4	European DLion keyboard
jLevel4	Japanese DLion keyboard
level5	matches any of the Daybreak keyboards (American, European or Japanese)
aLevel5	American Daybreak keyboard
eLevel5	European Daybreak keyboard
jLevel5	Japanese Daybreak keyboard
cursorKeys	keypad on Daybreak or sideKeys on Dlion is producing cursorKeys

A.4 Tables

A.4.1 Normal Tables

The set of Normal tables (**NormalMouse.TIP, NormalSoftKeys.TIP, NormalKeyboard.TIP, NormalSideKeys.TIP, NormalBackstop.TIP**) are registered at startup and pushed into the list of TIP tables at the appropriate **TIPStar** placeholder (**mouseActions, softKeys, blackKeys, sideKeys, backstopSpecialFocus**). (See **TIPStar** for further explanation about placeholders.) The set of Normal tables provides productions for all possible user input. Table entries are divided up into logical groups corresponding to the placeholder the table will be pushed onto. Thus input actions pertaining to the mouse (Point Down, Adjust Down, etc.) appear in the **NormalMouse.TIP** table and **NormalMouse.TIP** is pushed onto the **mouseActions** placeholder. Key actions from the side function key group that are directed at the input focus (MOVE Down, COPY Down, etc.) appear in the **NormalSideKeys.TIP** table and are pushed onto the **sideKeys** placeholder. Key actions such as the alphanumeric keys (A Down, 3 Down, etc.) appear in the **NormalKeyboard.TIP** table and are pushed onto the **blackKeys** placeholder. Key actions pertaining to the row of function keys at the top of the keyboard (CENTER Down, BOLD Down, etc.) appear in the **NormalSoftKeys.TIP** table and are pushed onto the **softKeys** placeholder. Key actions from the side function key group that are not directed at the input focus (KEYBOARD Down, HELP Down) appear in the **NormalBackstop.TIP** table and are pushed onto the **backstopSpecialFocus** placeholder.

At the end of ViewPoint boot sequence, the list of **TIP.Tables** in ViewPoint will appear as in Figure A-1.

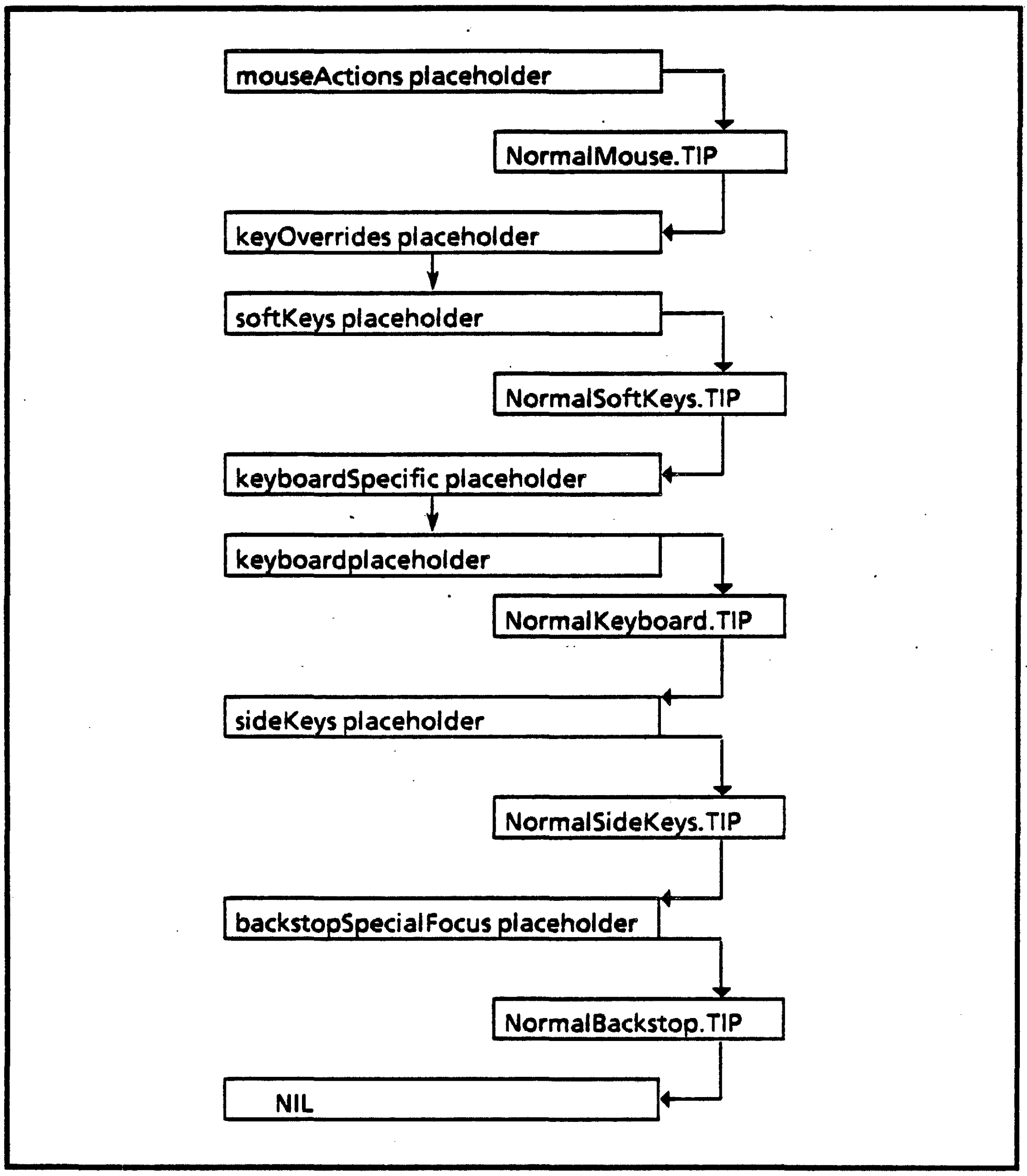


Figure A.1 TIP Tables after boot

--File: NormalBackstop.TIP *last edit: 14-Aug-86 11:50:24*

```
[DEF,IfShift,(SELECT ENABLE FROM
  LeftShift Down => ˜1;
  RightShift Down => ˜1;
  Key47 Down => ˜1;  -- JLevelIV keyboard LeftShiftAlt
  A12 Down => ˜1;   -- JLevelIV keyboard RightShiftAlt
  ENDCASE => ˜2)]

SELECT TRIGGER FROM

 FONT Down WHILE level4 => [IfShift,ShiftFontDown,FontDown];
 FONT Up WHILE level4 => [IfShift,ShiftFontUp,FontUp];
 KEYBOARD Down => [IfShift,ShiftKeyboardDown,KeyboardDown];
 KEYBOARD Up => [IfShift,ShiftKeyboardUp,KeyboardUp];
 HELP Down WHILE level4 => [IfShift,ShiftHelpDown,HelpDown];
 HELP Up WHILE level4 => [IfShift,ShiftHelpUp,HelpUp];
 STOP Down => [IfShift,ShiftStop,Stop];
 STOP Up => [IfShift,ShiftStopUp,StopUp];
 UNDO Down => [IfShift,ShiftUndoDown,UndoDown];
 UNDO Up => [IfShift,ShiftUndoUp,UndoUp];

 ENDCASE...
```

```
--File: NormalKeyboard.TIP last edit: 4-Mar-86 18:38:49

[DEF,IfShift,(SELECT ENABLE FROM
 LeftShift Down => ~1;
 RightShift Down => ~1;
 Key47 Down => SELECT ENABLE FROM
  jLevel4 => ~1; -- JLevelIV keyboard
  jLevel5 => ~1; -- JLevelV keyboard
 A12 Down => ~1; -- JLevelIV keyboard
ENDCASE => ~2)]

SELECT TRIGGER FROM

BS Down => [IfShift, BackWord, BackSpace];
Return Down => [IfShift, NewLine, NewParagraph];

Bullet Down => BUFFEREDCHAR;   --levelV  (non-existent on levelIV keyboard)
SingleQuote Down => BUFFEREDCHAR; --levelV  (non-existent on levelIV keyboard)

-- use predicates to distinguish physical keyboards
Key47 Down => SELECT ENABLE FROM --LeftShiftAlt in levelV terminology
 eLevel4 => BUFFEREDCHAR;    --European keyboard uses this as char key
 eLevel5 => BUFFEREDCHAR;    --Japanese keyboard uses this as shift key
 ENDCASE => LeftShiftAltDown;  --The key is non-existent on Amer keyboard

Zero Down => BUFFEREDCHAR;
One Down => BUFFEREDCHAR;
Two Down => BUFFEREDCHAR;
Three Down => BUFFEREDCHAR;
Four Down => BUFFEREDCHAR;
Five Down => BUFFEREDCHAR;
Six Down => BUFFEREDCHAR;
Seven Down => BUFFEREDCHAR;
Eight Down => BUFFEREDCHAR;
Nine Down => BUFFEREDCHAR;

A Down => BUFFEREDCHAR;
B Down => BUFFEREDCHAR;
C Down => BUFFEREDCHAR;
D Down => BUFFEREDCHAR;
E Down => BUFFEREDCHAR;
F Down => BUFFEREDCHAR;
G Down => BUFFEREDCHAR;
H Down => BUFFEREDCHAR;
I Down => BUFFEREDCHAR;
J Down => BUFFEREDCHAR;
K Down => BUFFEREDCHAR;
L Down => BUFFEREDCHAR;
M Down => BUFFEREDCHAR;
N Down => BUFFEREDCHAR;
O Down => BUFFEREDCHAR;
P Down => BUFFEREDCHAR;
```

```
Q Down = > BUFFEREDCHAR;
R Down = > BUFFEREDCHAR;
S Down = > BUFFEREDCHAR;
T Down = > BUFFEREDCHAR;
U Down = > BUFFEREDCHAR;
V Down = > BUFFEREDCHAR;
W Down = > BUFFEREDCHAR;
X Down = > BUFFEREDCHAR;
Y Down = > BUFFEREDCHAR;
Z Down = > BUFFEREDCHAR;

CloseQuote Down = > BUFFEREDCHAR;
DoubleQuote Down = > BUFFEREDCHAR; --levelV (A10 was unused on levelIV)
Comma Down = > BUFFEREDCHAR;
Minus Down = > BUFFEREDCHAR;
Equal Down = > BUFFEREDCHAR;
LeftBracket Down = > BUFFEREDCHAR;
Period Down = > BUFFEREDCHAR;
OpenQuote Down = > BUFFEREDCHAR;
RightBracket Down = > BUFFEREDCHAR;
SemiColon Down = > BUFFEREDCHAR;
Space Down = > BUFFEREDCHAR;
Slash Down = > BUFFEREDCHAR;

-- use predicates to distinguish physical keyboards
PARATAB Down = > SELECT ENABLE FROM
 eLevel4 = > [IfShift, TabDown, ParaTabDown];
 level5 = > [IfShift, TabDown, ParaTabDown];
 ENDCASE = > ParaTabDown;
TAB Down = > SELECT ENABLE FROM
 eLevel4 = > BUFFEREDCHAR;
 level5 = > BUFFEREDCHAR;
 ENDCASE = >TabDown;

LOCK Down = > LockDown;
LOCK Up = > LockUp;

-- JStar keyboards
-- Note: A8, A9, A11, A12 and Key47 exist only on the J keyboards.
A11 Down = > BUFFEREDCHAR;
A8 Down = > BUFFEREDCHAR;
A9 Down = > BUFFEREDCHAR;

-- Diagnostics bits
DiagnosticBitOne Down = > DiagnosticBitOneDown; --levelV (D1 was unused on levelIV)
DiagnosticBitTwo Down = > DiagnosticBitTwoDown; --levelV (D2 was unused on levelIV)

ENDCASE..
```

```
--File: NormalMouse.TIP last edit: 3-Apr-86 15:29:19

OPTIONS Small;

[DEF,SHIFT,(LeftShift Down | RightShift Down | Key47 Down | A12 Down)]
[DEF,ChordP,(SELECT TRIGGER FROM
 ¯1 Down BEFORE 200 => {TIME COORDS Menu PointDown AdjustDown};
 ENDCASE => ¯2 )]

[DEF,ChordA,(SELECT TRIGGER FROM
 ¯1 Down BEFORE 200 => {TIME COORDS Menu AdjustDown PointDown};
 ENDCASE => ¯2 )]

SELECT TRIGGER FROM

 MOUSE => SELECT ENABLE FROM
  Point Down => COORDS, PointMotion;
  Adjust Down => COORDS, AdjustMotion;
  MouseMiddle Down => COORDS, MouseMiddleMotion;
 ENDCASE;

 Point Down => [ChordP,Adjust,
  SELECT ENABLE FROM
   [SHIFT] => {TIME COORDS Menu PointDown};
  ENDCASE => {TIME COORDS PointDown}];
 Point Up =>
  SELECT ENABLE FROM
   [SHIFT] => TIME, COORDS, Shift, PointUp;
  ENDCASE => TIME, COORDS, PointUp;

 Adjust Down => [ChordA,Point,
  SELECT ENABLE FROM
   [SHIFT] => {TIME COORDS Menu AdjustDown};
  ENDCASE => {TIME COORDS AdjustDown}];
 Adjust Up =>
  SELECT ENABLE FROM
   [SHIFT] => TIME, COORDS, Shift, AdjustUp;
  ENDCASE => TIME, COORDS, AdjustUp;

 MouseMiddle Down => SELECT ENABLE FROM
  [SHIFT] => TIME, COORDS, Shift, MouseMiddleDown;
 ENDCASE => TIME, COORDS, MouseMiddleDown;

 MouseMiddle Up => SELECT ENABLE FROM
  [SHIFT] => TIME, COORDS, Shift, MouseMiddleUp;
 ENDCASE => TIME, COORDS, MouseMiddleUp;

 ENTER => Enter;
 EXIT => Exit;

ENDCASE..
```

```
--File: NormalSideKeys.TIP last edit: 24-Apr-87 13:37:50

[DEF,IfShift,(SELECT ENABLE FROM
  LeftShift Down => ~1;
  RightShift Down => ~1;
  Key47 Down => ~1;  -- JLevelIV keyboard LeftShiftAlt
  A12 Down => ~1;    -- JLevelIV keyboard RightShiftAlt
  ENDCASE => ~2)]


SELECT TRIGGER FROM

-- left function keys on both daybreak and dlion keyboards
 AGAIN Down => [IfShift,ShiftAgainDown,AgainDown];
 AGAIN Up => [IfShift,ShiftAgainUp,AgainUp];
 DELETE Down => [IfShift,ShiftDeleteDown,DeleteDown];
 DELETE Up => [IfShift,ShiftDeleteUp,DeleteUp];
 FIND Down => [IfShift,ShiftFindDown,FindDown];
 FIND Up => [IfShift,ShiftFindUp,FindUp];
 COPY Down => [IfShift,ShiftCopyDown,CopyDown];
 COPY Up => [IfShift,ShiftCopyUp,CopyUp];
 SAME Down => [IfShift,ShiftSameDown,SameDown];
 SAME Up => [IfShift,ShiftSameUp,SameUp];
 MOVE Down => [IfShift,ShiftMoveDown,MoveDown];
 MOVE Up => [IfShift,ShiftMoveUp,MoveUp];
 OPEN Down => [IfShift,ShiftOpenDown,OpenDown];
 OPEN Up => [IfShift,ShiftOpenUp,OpenUp];
 PROPS Down => [IfShift,ShiftPropsDown,PropsDown];
 PROPS Up => [IfShift,ShiftPropsUp,PropsUp];

-- beside space bar on daybreak and in right function group on dlion
 EXPAND Down => [IfShift,DefineDown,ExpandDown];
 EXPAND Up => [IfShift,DefineUp,ExpandUp];

-- right function keys on both keyboards
 NEXT Down => SELECT ENABLE FROM
  cursorKeys => SELECT ENABLE FROM
   level4 => [IfShift,HomeDown,LeftArrowDown];
   ENDCASE => [IfShift,SkipDown,NextDown];
  ENDCASE => [IfShift,SkipDown,NextDown];
 NEXT Up => SELECT ENABLE FROM
  cursorKeys => SELECT ENABLE FROM
   level4 => [IfShift,HomeUp,LeftArrowUp];
   ENDCASE => [IfShift,SkipUp,NextUp];
  ENDCASE => [IfShift,SkipUp,NextUp];

-- part of the right function group on dlion but moved to softkeys on daybreak
 MARGINS Down WHILE level4 => SELECT ENABLE FROM
  cursorKeys => [IfShift,EndPageDown,DwnArrowDown];
   ENDCASE => [IfShift,ShiftMarginsDown,MarginsDown];
```

```
MARGINS Up WHILE level4 => SELECT ENABLE FROM
 cursorKeys => [IfShift,EndPageUp,DwnArrowUp];
 ENDCASE => [IfShift,ShiftMarginsUp,MarginsUp];

-- Added for Cursor Keys
HELP Down WHILE cursorKeys => [IfShift,StartPageDown,UpArrowDown];
HELP Up WHILE cursorKeys => [IfShift,StartPageUp,UpArrowUp];
UNDO Down WHILE level4 => SELECT ENABLE FROM
 cursorKeys => [IfShift,EndDown,RightArrowDown];
 ENDCASE;
UNDO Up WHILE level4 => SELECT ENABLE FROM
 cursorKeys => [IfShift,EndUp,RightArrowUp];
 ENDCASE;


-- calculator key pad on daybreak
-- (has no meaning on dlion except in virtual keypad)
KeypadZero Down => BUFFEREDCHAR;

KeypadOne Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
  cursorKeys => [IfShift,BUFFEREDCHAR,EndDown];
  ENDCASE => [IfShift,EndDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadOne Up => EndUp;

KeypadTwo Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
  cursorKeys => [IfShift,BUFFEREDCHAR,DwnArrowDown];
  ENDCASE => [IfShift,DwnArrowDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadTwo Up => DwnArrowUp;

KeypadThree Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
  cursorKeys => [IfShift,BUFFEREDCHAR,NextPageDown];
  ENDCASE => [IfShift,NextPageDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadThree Up => NextPageUp;

KeypadFour Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
  cursorKeys => [IfShift,BUFFEREDCHAR,LeftArrowDown];
  ENDCASE => [IfShift,LeftArrowDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadFour Up => LeftArrowUp;

KeypadFive Down => BUFFEREDCHAR;
```

```
KeypadSix Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
  cursorKeys => [IfShift,BUFFEREDCHAR,RightArrowDown];
  ENDCASE => [IfShift,RightArrowDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadSix Up => RightArrowUp;

KeypadSeven Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
  cursorKeys => [IfShift,BUFFEREDCHAR,HomeDown];
  ENDCASE => [IfShift,HomeDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadSeven Up => HomeUp;

KeypadEight Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
  cursorKeys => [IfShift,BUFFEREDCHAR,UpArrowDown];
  ENDCASE => [IfShift,UpArrowDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadEight Up => UpArrowUp;

KeypadNine Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
 cursorKeys => [IfShift,BUFFEREDCHAR,PrevPageDown];
 ENDCASE => [IfShift,PrevPageDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadNine Up => PrevPageUp;

KeypadAdd Down => BUFFEREDCHAR;
KeypadSubtract Down => BUFFEREDCHAR;
KeypadMultiply Down => BUFFEREDCHAR;
KeypadDivide Down => BUFFEREDCHAR;

KeypadPeriod Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
  cursorKeys => [IfShift,BUFFEREDCHAR,StartPageDown];
  ENDCASE => [IfShift,StartPageDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadPeriod Up => StartPageUp;

KeypadComma Down => SELECT ENABLE FROM
 level5 => SELECT ENABLE FROM
  cursorKeys => [IfShift,BUFFEREDCHAR,EndPageDown];
  ENDCASE => [IfShift,EndPageDown,BUFFEREDCHAR];
 ENDCASE => BUFFEREDCHAR;
KeypadComma Up => EndPageUp;

KeypadClear Down => [IfShift,ShiftClearDown,ClearDown];
KeypadClear Up => [IfShift,ShiftClearUp,ClearUp];

ENDCASE...
```

```
-- File: NormalSoftKeys.TIP  last edit: 5-Mar-86 12:40:42

-- SoftKeys are the top row of function keys
[DEF,IfShift,(SELECT ENABLE FROM
  LeftShift Down => ~1;
  RightShift Down => ~1;
  Key47 Down => ~1; -- JLevelIV keyboard LeftShiftAlt
  A12 Down => ~1;   -- JLevelIV keyboard RightShiftAlt
  ENDCASE => ~2))]


SELECT TRIGGER FROM
-- top function keys
 CENTER Down => [IfShift,ShiftCenterDown,CenterDown];
 CENTER Up => [IfShift,ShiftCenterUp,CenterUp];
 BOLD Down => [IfShift,UnboldDown,BoldDown];
 BOLD Up => [IfShift,UnboldUp,BoldUp];
 ITALICS Down => [IfShift,ShiftItalicsDown,ItalicsDown];
 ITALICS Up => [IfShift,ShiftItalicsUp,ItalicsUp];

-- Case key on daybreak only
 Case Down => [IfShift,ShiftCaseDown,CaseDown];
 Case Up => [IfShift,ShiftCaseUp,CaseUp];

 UNDERLINE Down => SELECT ENABLE FROM
  level4 => [IfShift,ShiftUnderlineDown,UnderlineDown];
  level5 => [IfShift,ShiftDbkUnderlineDown,DbkUnderlineDown];
  ENDCASE => [IfShift,ShiftUnderlineDown,UnderlineDown];
 UNDERLINE Up => SELECT ENABLE FROM
  level4 => [IfShift,ShiftUnderlineUp,UnderlineUp];
  level5 => [IfShift,ShiftDbkUnderlineUp,DbkUnderlineUp];
  ENDCASE => [IfShift,ShiftUnderlineUp,UnderlineUp];

--strikeout and supersub on daybreak only
 Strikeout Down => [IfShift,ShiftStrikeoutDown,StrikeoutDown];
 Strikeout Up => [IfShift,ShiftStrikeoutUp,StrikeoutUp];
 SuperSub Down => [IfShift,ShiftSuperSubDown,SuperSubDown];
 SuperSub Up => [IfShift,ShiftSuperSubUp,SuperSubUp];

--superscript and subscript on dlion only
 SUPERSCRIPT Down => [IfShift,ShiftSuperscriptDown,SuperscriptDown];
 SUPERSCRIPT Up => [IfShift,ShiftSuperscriptUp,SuperscriptUp];
 SUBSCRIPT Down => [IfShift,ShiftSubscriptDown,SubscriptDown];
 SUBSCRIPT Up => [IfShift,ShiftSubscriptUp,SubscriptUp];

 SMALLER Down => SELECT ENABLE FROM
  level4 => [IfShift,LargerDown,SmallerDown];
  level5 => [IfShift,DbkLargerDown,DbkSmallerDown];
  ENDCASE => [IfShift,LargerDown,SmallerDown];
```

```
SMALLER Up = > SELECT ENABLE FROM
 level4 = > [IfShift,LargerUp,SmallerUp];
 level5 = > [IfShift,DbkLargerUp,DbkSmallerUp];
 ENDCASE = > [IfShift,LargerUp,SmallerUp];

--margins key is a softkey on daybreak and a right function key on dlion
MARGINS Down WHILE level5 = > [IfShift,ShiftMarginsDown,MarginsDown];
MARGINS Up WHILE level5 = > [IfShift,ShiftMarginsUp,MarginsUp];

--defaults key on dlion only
DEFAULTS Down = > [IfShift,ShiftDefaultsDown,DefaultsDown];
DEFAULTS Up = > [IfShift,ShiftDefaultsUp,DefaultsUp];

--font key is a softkey on daybreak and a right function key on dlion
FONT Down WHILE level5 = > [IfShift,ShiftFontDown,FontDown];
FONT Up WHILE level5 = > [IfShift,ShiftFontUp,FontUp];

ENDCASE...
```

A.4.2 Mouse Mode Tables

The mouse mode tables refer to the set of tables that will be swapped in and out of the **TIPStar** watershed at the **mouseActions** placeholder, depending on the mode set in **TIPStar.SetMode**. **TIPStar.Modes** are **normal, copy, move,** and **sameAs**.

Note: **mode = normal** will return **NormalMouse.TIP** to the watershed.

```
--File: CopyModeMouse.TIP last edit: 20-Apr-87 16:35:27

OPTIONS Small;

SELECT TRIGGER FROM

 MOUSE = > SELECT ENABLE FROM
  Point Down = > COORDS, CopyModeMotion,MouseLeft;
  Adjust Down = > COORDS, CopyModeMotion,MouseRight;
 ENDCASE;
 Point Down = > COORDS, CopyModeDown, KEY;
 Point Up = > COORDS, CopyModeUp, KEY;
 Adjust Down = > COORDS, CopyModeDown, KEY;
 Adjust Up = > COORDS, CopyModeUp, KEY;

ENTER = > CopyModeEnter;
EXIT = > CopyModeExit;

ENDCASE..
```

```
--File: MoveModeMouse.TIP last edit: 20-Apr-87 16:35:21

OPTIONS Small;

SELECT TRIGGER FROM

 MOUSE = > SELECT ENABLE FROM
  Point Down = > COORDS, MoveModeMotion,MouseLeft;
  Adjust Down = > COORDS, MoveModeMotion,MouseRight;
  ENDCASE;

 Point Down = > COORDS, MoveModeDown, KEY;
 Point Up = > COORDS, MoveModeUp, KEY;
 Adjust Down = > COORDS, MoveModeDown, KEY;
 Adjust Up = > COORDS, MoveModeUp, KEY;

ENTER = > MoveModeEnter;
EXIT = > MoveModeExit;

 ENDCASE..
```

--File: SameAsModeMouse.TIP *last edit: 20-Apr-87 16:35:58*

```
OPTIONS Small;

SELECT TRIGGER FROM

 MOUSE = > SELECT ENABLE FROM
  Point Down = > COORDS, SameAsModeMotion,MouseLeft;
  Adjust Down = > COORDS, SameAsModeMotion,MouseRight;
 ENDCASE;

 Point Down = > COORDS, SameAsModeDown, KEY;
 Point Up = > COORDS, SameAsModeUp, KEY;
 Adjust Down = > COORDS, SameAsModeDown, KEY;
 Adjust Up = > COORDS, SameAsModeUp, KEY;

ENTER = > SameAsModeEnter;
EXIT = > SameAsModeExit;

ENDCASE...
```

A.5 Usage/Examples

A.5.1 Using NormalSoftKeys.TIP when installing client softKeys

```
-- define the Atoms for my NotifyProc to use --
centerDown, boldDown, italicsDown, underlineDown, superscriptDown,
  subscriptDown, smallerDown, defaultsDown : Atom.ATOM ← Atom.null;

Init: PROCEDURE =
BEGIN
 -- initialize my Atoms --
   centerDown ← Atom.MakeAtom["CenterDown"];
   boldDown ← Atom.MakeAtom["BoldDown"L];
   italicsDown ← Atom.MakeAtom["ItalicsDown"L];
   underlineDown ← Atom.MakeAtom["UnderlineDown"L];
   superscriptDown ← Atom.MakeAtom["SuperscriptDown"L];
   subscriptDown ← Atom.MakeAtom["SubscriptDown"L];
   smallerDown ← Atom.MakeAtom["SmallerDown"L];
END; --Init

-- somewhere in the code --
 softKeyHandle ← SoftKeys.Push[
   notifyProc: MyNotifyProc,
   labels: DESCRIPTOR[labels, SoftKeys.numberOfKeys] ];

MyNotifyProc: TIP.NotifyProc =
  BEGIN
   FOR input: TIP.Results ← results, input.next UNTIL input = NIL DO
     WITH z: input SELECT FROM
     atom => SELECT z.a FROM
       centerDown => --Do something interesting--;
       boldDown => --Do something interesting--;
       italicsDown => --Do something interesting--;
       underlineDown => --Do something interesting--;
       superscriptDown => --Do something interesting--;
       subscriptDown => --Do something interesting--;
       smallerDown => --Do something interesting--;
       defaultsDown => --Do something interesting--;

       ENDCASE
     ENDCASE
   ENDLOOP
  END; -- MyNotifyProc
```

MyNotifyProc will be attached to NormalSoftKeys.TIP by the SoftKeys implementation. Until this client does a SoftKeys.Remove, whenever the user presses one of the top row function keys MyNotifyProc will be called with the appropriate production from the NormalSoftKeys.TIP.

A.5.2 Attaching a NotifyProc to One of the Normal Tables

If a client application wants to grab the use, for example, of all the side keys for some period of time, it can attach a **notifyProc** to the **NormalSideKeys.TIP** table by calling:

```
old ← TIP.SetNotifyProcForTable[TIPStar.GetTable[sideKeys], MyNotifyProc];
```

A.6 Index of TIP Tables

Appendix B

References

The following documents should be studied before or in conjunction with this manual:

- *Mesa Language Manual* (610E00170).
- *XDE User Guide* (610E00140).
- *Pilot Programmer's Manual* (610E00160).
- *Srvices Programmer's Guide: Filing Programmer's Manual* (610E00180).

In addition, any other documentation accompanying a release of ViewPoint should be consulted before writing any programs. A list of this documentation can be found in the release message for each release.

Appendix C

Listing of Atoms

C.1 Overview

Atoms (see the **Atom** interface) are used in several places in ViewPoint. This appendix contains a list of the strings that represent them.

C.2 Atoms as TIP Results in the System TIP Tables

Most of the right-hand sides (TIP results) of the productions in the system-provided TIP Tables (see Appendix A) contain atoms.

AdjustDown
AdjustMotion
AdjustUp
AgainDown
AgainUp
aLevel4
aLevel5
BackSpace
BackWord
BoldDown
BoldUp
CaseDown
CaseUp
CenterDown
CenterUp
ClearDown
ClearUp
CopyDown
CopyModeDown
CopyModeEnter
CopyModeExit
CopyModeMotion
CopyModeMouse
CopyModeUp
CopyUp
cursorKeys

DbkLargerDown
DbkLargerUp
DbkSmallerDown
DbkSmallerUp
DbkUnderlineDown
DbkUnderlineUp
DefaultsDown
DefaultsUp
DefineDown
DefineUp
DeleteDown
DeleteUp
DiagnosticBitOne
DiagnosticBitOneDown
DiagnosticBitTwo
DiagnosticBitTwoDown
DwnArrowDown
DwnArrowUp
eLevel4
eLevel5
EndDown
EndPageDown
EndPageUp
EndUp
Enter
Exit
ExpandDown
ExpandUp
FindDown
FindUp
FontDown
FontUp
HelpDown
HelpUp
HomeDown
HomeUp
ItalicsDown
ItalicsUp
jLevel4
jLevel5
KeyboardDown
KeyboardUp
KeypadAdd
KeypadClear
KeypadComma
KeypadDivide
KeypadEight
KeypadFive
KeypadFour
KeypadMultiply
KeypadNine
KeypadOne

KeypadPeriod
KeypadSeven
KeypadSix
KeypadSubtract
KeypadThree
KeypadTwo
KeypadZero
LargerDown
LargerUp
LeftArrowDown
LeftArrowUp
LeftShiftAltDown
level4
level5
LockDown
LockUp
MarginsDown
MarginsUp
MouseMiddleDown
MouseMiddleMotion
MouseMiddleUp
MoveDown
MoveModeDown
MoveModeEnter
MoveModeExit
MoveModeMotion
MoveModeMouse
MoveModeUp
NewLine
NewParagraph
NextDown
NextPageDown
NextPageUp
NextUp
NumLockKeyDown
OpenDown
OpenQuote
OpenUp
ParaTabDown
PointDown
PointMotion
PointUp
PrevPageDown
PrevPageUp
PropsDown
PropsUp
RightArrowDown
RightArrowUp
SameAsModeDown
SameAsModeEnter
SameAsModeExit
SameAsModeMotion

SameAsModeMouse
SameAsModeUp
SameDown
SameUp
ShiftAgainDown
ShiftAgainUp
ShiftCaseDown
ShiftCaseUp
ShiftCenterDown
ShiftCenterUp
ShiftClearDown
ShiftClearUp
ShiftCopyDown
ShiftCopyUp
ShiftDbkUnderlineDown
ShiftDbkUnderlineUp
ShiftDefaultsDown
ShiftDefaultsUp
ShiftDeleteDown
ShiftDeleteUp
ShiftFindDown
ShiftFindUp
ShiftFontDown
ShiftFontUp
ShiftHelpDown
ShiftHelpUp
ShiftItalicsDown
ShiftItalicsUp
ShiftKeyboardDown
ShiftKeyboardUp
ShiftMarginsDown
ShiftMarginsUp
ShiftMoveDown
ShiftMoveUp
ShiftOpenDown
ShiftOpenUp
ShiftPropsDown
ShiftPropsUp
ShiftSameDown
ShiftSameUp
ShiftStop
ShiftStopUp
ShiftStrikeoutDown
ShiftStrikeoutUp
ShiftSubscriptDown
ShiftSubscriptUp
ShiftSuperscriptDown
ShiftSuperscriptUp
ShiftSuperSubDown
ShiftSuperSubUp
ShiftUnderlineDown
ShiftUnderlineUp

ShiftUndoDown
ShiftUndoUp
SkipDown
SkipUp
SmallerDown
SmallerUp
StartPageDown
StartPageUp
Stop
StopUp
Strikeout
StrikeoutDown
StrikeoutUp
subscript
SubscriptDown
SubscriptUp
SuperscriptDown
SuperscriptUp
SuperSubDown
SuperSubUp
TabDown
UnboldDown
UnboldUp
UnderlineDown
UnderlineUp
UndoDown
UndoUp
UpArrowDown
UpArrowUp

C.3 Passed as the "Atom" Parameter to a Containee.GenericProc

These atoms may be passed to a **Containee.GenericProc** as the **atom** parameter, indicating what operation the **GenericProc** should perform:

CanYouTakeSelection
CanYouTakeSelectionBackground
CanYouTakeSelectionAndFork
FreeMenu
Menu
Open
Props
TakeSelection
TakeSelectionAndFork
TakeSelectionBackground
TakeSelectionCopy
TakeSelectionCopyAndFork
TakeSelectionCopyBackground

C.4 Event Atoms

Events are identified by atoms (see the Event interface). The following events are explained in further detail in the chapter indicated.

Event	EventData	Chapter where discussed
AboutLoading	ApplicationFolderExtra.EventData	ApplicationFolder
ApplicationLoaded	ApplicationFolderExtra.EventData	ApplicationFolder
AtomicProfileChange	LPT Atom.ATOM*	AtomicProfile
AttemptingLogoff	LPT SpecialLogon.ActiveQueueRequest	StarDesktop
AttemptingLogoffFailed	NIL	StarDesktop
BlackKeysChange	BlackKeys.Keyboard	BlackKeys
DesktopWindowAvailable	NIL	StarDesktop
LoadedAndAboutToStart	ApplicationFolderExtra.EventData	ApplicationFolder
LoadVetoed	ApplicationFolderExtra.EventData	ApplicationFolder
NewIcon	NIL	StarDesktop
NewImplementation	LPT NSFile.Type	Containee
Logoff	NIL	StarDesktop
Logon	NIL	StarDesktop
LogonCompleted	NIL	StarDesktop
NewSystemFont	SimpleTextFont.MappedFontHandle	SimpleTextFont
NewSystemFontHeight	LPT CARDINAL	SimpleTextDisplay
PlainTextFileEdited	LPT NSFile.Reference	**

*LPT stands for LONG POINTER TO
**PlainTextFileEdited is notified bySimpleEditorImpl when a plain text file is edited.

C.5 AtomicProfile Atoms

AtomicProfile is used to save various values globally. Values are saved with the following atoms. See the AtomicProfile Chapter for information on how to retrieve the associated value.

Atom: FullUserName - Associated Value: An XString.Reader, the fully qualified user's name as entered by the user at logon.

C.6 Other

The **Atom** interface allows any value to be associated with any pair of atoms (see **Atom.GetProp**, **Atom.Pair**, etc.).

Atom	Property atom	Value
CurrentUser	ConversationHandle	LONG POINTER TO CH.ConversationHandle created by Logon during user authentication.
	FileService	The NSFile.Service for the user's home file service.
	FullUserName	This XString.Reader is the fully qualified name of the logged on user.
	IdentityHandle	Auth.IdentityHandle for the currently logged on user. This is created at logon if the user enters a password; it can be used to access any Network Service. It is created with strong authentication.
	NSName	This is an NSName.Name for the fully qualified name of the logged on user
	SimpleIdentityHandle	Just like IdentityHandle above, but created with simple authentication.
MultiNational	ExtendedLanguage	LONG POINTER TO BOOLEAN
	Language	LPT MultiNational.Language
	PaperSizes	LPT MultiNational.PageSizes
	PhysicalKeyboard	LPT MultiNational.PhysicalKeyboard
	SortOrder	LPT XString.SortOrder
	Units	LPT MultiNational.Unit

These MultiNational values are obtained from the WorkstationProfile. They are intended to be used for customizing workstations for a particular country. MultiNational.mesa is a Friends interface.

Appendix D

Listing of Public Symbols

This appendix lists all public items from the public interfaces, i.e., the files in **XStringPublic.df** and **BWSPublic.df**.

-- *ApplicationFolder Atom AtomicProfile Attention BlackKeys BWSAttributeTypes BWSFileTypes BWSZone Catalog Containee ContainerCache ContainerCacheExtra ContainerSource ContainerWindow ContainerWindowExtra ContainerWindowExtra2 Context Cursor Display Event FileContainerShell FileContainerSource FileContainerSourceExtra FileContainerSourceExtra2 FormWindow FormWindowMessageParse IdleControl KeyboardKey KeyboardWindow LevelIVKeys MenuData MessageWindow OptionFile PopupMenu ProductFactoring ProductFactoringProducts ProductFactoringProductsExtras PropertySheet Prototype PrototypeExtra Selection SimpleTextDisplay SimpleTextEdit SimpleTextFont SimpleTextFontExtra SoftKeys StarDesktop StarWindowShell StarWindowShellExtra StarWindowShellExtra2 TIP TIPStar TIPX Undo UnitConversion Window XChar XCharSet0 XCharSet164 XCharSet356 XCharSet357 XCharSet360 XCharSet361 XCharSet41 XCharSet42 XCharSet43 XCharSet44 XCharSet45 XCharSet46 XCharSet47 XCharSets XComSoftMessage XFormat XLReal XMessage XString XTime XToken*

Abs: --*XLReal*-- PROCEDURE [Number] RETURNS [Number];
accentedLatin: --*XCharSet361*-- XCharSets.Sets = LOOPHOLE[241];
accuracy: --*XLReal*-- NATURAL = 13;
Acquire: --*Context*-- PROCEDURE [type: Type, window: Window.Handle]
  RETURNS [Data];
Action: --*ContainerSource*-- TYPE = {destroy, reList, sleep, wakeup};
Action: --*Selection*-- TYPE = MACHINE DEPENDENT{
  clear, mark, unmark, delete, clearIfHasInsert, save, restore, firstFree,
  last(255)};
actionToWindow: --*TIP*-- PACKED ARRAY KeyName OF BOOLEAN;
ActOn: --*ContainerSource*-- ActOnProc;
ActOn: --*Selection*-- PROCEDURE [action: Action];
ActOnProc: --*ContainerSource*-- TYPE = PROCEDURE [
  source: Handle, action: Action];
ActOnProc: --*Selection*-- TYPE = PROCEDURE [data: ManagerData, action:
Action]
  RETURNS [cleared: BOOLEAN ←FALSE];

Add: --*PrototypeExtra*-- PROCEDURE [
  file: NSFile.Handle, version: Prototype.Version,
  subtype: Prototype.Subtype ^0, session: NSFile.Session ^LOOPHOLE[0]];
Add: --*XLReal*-- PROCEDURE [a: Number, b: Number] RETURNS [Number];
AddClientDefinedCharacter: --*SimpleTextFont*-- PROCEDURE [
  width: CARDINAL, height: CARDINAL, bitsPerLine: CARDINAL, bits: LONG POINTER,
  offsetIntoBits: CARDINAL ^0] RETURNS [XString.Character];
AddData: --*ContainerCache*-- TYPE = RECORD [
  clientData: LONG POINTER,
  clientDataCount: CARDINAL,
  clientStrings: LONG DESCRIPTOR FOR ARRAY CARDINAL OF XString.ReaderBody];
AddDependencies: --*Event*-- PROCEDURE [
  agent: AgentProcedure, myData: LONG POINTER,
  events: LONG DESCRIPTOR FOR ARRAY CARDINAL OF EventType,
  remove: FreeDataProcedure ^NIL] RETURNS [dependency: Dependency];
AddDependency: --*Event*-- PROCEDURE [
  agent: AgentProcedure, myData: LONG POINTER, event: EventType,
  remove: FreeDataProcedure ^NIL] RETURNS [dependency: Dependency];
AddItem: --*MenuData*-- PROCEDURE [menu: MenuHandle, new: ItemHandle];
AddMenuItem: --*Attention*-- PROCEDURE [item: MenuData.ItemHandle];
AddPopupMenu: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, menu: MenuData.MenuHandle];
AddReferenceToDesktop: --*StarDesktop*-- PROCEDURE [
  reference: NSFile.Reference, place: Window.Place ^nextPlace];
AddToSystemKeyboards: --*KeyboardKey*-- PROCEDURE [keyboard:
BlackKeys.Keyboard];
AdjustProc: --*StarWindowShell*-- TYPE = PROCEDURE [
  sws: Handle, box: Window.Box, when: When];
AgentProcedure: --*Event*-- TYPE = PROCEDURE [
  event: EventType, eventData: LONG POINTER, myData: LONG POINTER]
  RETURNS [remove: BOOLEAN ^FALSE, veto: BOOLEAN ^FALSE];
AllocateAndInsert: --*MessageWindow*-- PROCEDURE [
  parent: Window.Handle, place: Window.Place ^LOOPHOLE[0],
  dims: Window.Dims ^LOOPHOLE[23417B], zone: UNCOUNTED ZONE ^LOOPHOLE[0],
  lines: CARDINAL ^10] RETURNS [Window.Handle];
AllocateCache: --*ContainerCache*-- PROCEDURE RETURNS [Handle];
AllocateMessages: --*XMessage*-- PROCEDURE [
  applicationName: LONG STRING, maxMessages: CARDINAL, clientData: ClientData,
  proc: DestroyMsgsProc] RETURNS [h: Handle];
Alphabetic: --*XToken*-- FilterProcType;
AlphaNumeric: --*XToken*-- FilterProcType;
Append: --*XTime*-- PROCEDURE [
  w: XString.Writer, time: System.GreenwichMeanTime ^defaultTime,
  template: XString.Reader ^dateAndTime, ltp: LTP ^useSystem];
AppendChar: --*XString*-- PROCEDURE [
  to: Writer, c: Character, extra: CARDINAL ^0];
AppendExtensionIfNeeded: --*XString*-- PROCEDURE [to: Writer, extension: Reader]
  RETURNS [didAppend: BOOLEAN];
AppendItem: --*ContainerCache*-- PROCEDURE [cache: Handle, addData: AddData]
  RETURNS [handle: ItemHandle];

```
AppendItem: --FormWindow-- PROCEDURE [
   window: Window.Handle, item: ItemKey, line: Line, preMargin: CARDINAL ^0,
   tabStop: CARDINAL ^nextTabStop, repaint: BOOLEAN ^TRUE];
AppendLine: --FormWindow-- PROCEDURE [
   window: Window.Handle, spaceAboveLine: CARDINAL ^0] RETURNS [line: Line];
AppendReader: --XString-- PROCEDURE [
   to: Writer, from: Reader, fromEndContext: Context ^unknownContext,
   extra: CARDINAL ^0];
AppendStream: --XString-- PROCEDURE [
   to: Writer, from: Stream.Handle, nBytes: CARDINAL,
   fromContext: Context ^vanillaContext, extra: CARDINAL ^0]
   RETURNS [bytesTransferred: CARDINAL];
AppendSTRING: --XString-- PROCEDURE [
   to: Writer, from: LONG STRING, homogeneous: BOOLEAN ^FALSE,
   extra: CARDINAL ^0];
ArabicFirstRightToLeftCharCode: --XChar-- Environment.Byte = 48;
Arc: --Display-- PROCEDURE [
   window: Handle, place: Window.Place, radius: INTEGER, startSector: CARDINAL,
   stopSector: CARDINAL, start: Window.Place, stop: Window.Place,
   lineStyle: LineStyle ^NIL, bounds: Window.BoxHandle ^NIL];
ArcCos: --XLReal-- PROCEDURE [x: Number] RETURNS [radians: Number];
ArcSin: --XLReal-- PROCEDURE [x: Number] RETURNS [radians: Number];
ArcTan: --XLReal-- PROCEDURE [x: Number] RETURNS [radians: Number];
ArrayHandle: --MenuData-- TYPE = LONG DESCRIPTOR FOR ARRAY CARDINAL OF
   ItemHandle;
ArrowFlavor: --StarWindowShell-- TYPE = {pageFwd, pageBwd, forward, backward};
ArrowScrollAction: --StarWindowShell-- TYPE = {start, go, stop};
ArrowScrollProc: --StarWindowShell-- TYPE = PROCEDURE [
   sws: Handle, vertical: BOOLEAN, flavor: ArrowFlavor,
   arrowScrollAction: ArrowScrollAction ^go];
ATOM: --Atom-- TYPE [1];
ATOM: --TIP-- TYPE = Atom.ATOM;
attemptingLogoff: --StarDesktop-- Atom.ATOM;
AttentionProc: --TIP-- TYPE = PROCEDURE [window: Window.Handle];
AttributeFormatProc: --FileContainerSource-- TYPE = PROCEDURE [
   containeeImpl: Containee.Implementation, containeeData: Containee.DataHandle,
   attr: NSFile.Attribute, displayString: XString.Writer];
BackScanClosure: --XString-- TYPE = RECORD [
   proc: BackScanProc, env: LONG POINTER];
BackScanProc: --XString-- TYPE = PROCEDURE [
   beforePos: CARDINAL, env: LONG POINTER]
   RETURNS [pos: CARDINAL, context: Context];
backStopInputFocus: --TIP-- READONLY Window.Handle;
beforeItemZero: --ContainerSource-- ItemIndex = 177776B;
beforeLogonSession: --Catalog-- NSFile.Session;
BeginFill: --ContainerCache-- PROCEDURE [
   cache: Handle, fillProc: FillProc, clients: LONG POINTER,
   fork: BOOLEAN ^TRUE];
Bit: --LevelIVKeys-- TYPE = KeyStations.Bit;
BitAddress: --Display-- TYPE = BitBlt.BitAddress;
```

```
BitAddressFromPlace: --Display-- PROCEDURE [
   base: BitAddress, x: NATURAL, y: NATURAL, raster: CARDINAL]
   RETURNS [BitAddress];
BitBltFlags: --Display-- TYPE = BitBlt.BitBltFlags;
bitFlags: --Display-- BitBltFlags;
Bitmap: --Display-- PROCEDURE [
   window: Handle, box: Window.Box, address: BitAddress,
   bitmapBitWidth: CARDINAL, flags: BitBltFlags ^paintFlags,
   bounds: Window.BoxHandle ^NIL];
Bitmap: --FormWindow-- TYPE = RECORD [
   height: CARDINAL,
   width: CARDINAL,
   bitsPerLine: CARDINAL,
   bits: Environment.BitAddress];
BitmapPlace: --Window-- PROCEDURE [window: Handle, place: Place ^LOOPHOLE[0]]
   RETURNS [Place];
BitmapPlaceToWindowAndPlace: --Window-- PROCEDURE [bitmapPlace: Place]
   RETURNS [window: Handle, place: Place];
Bits: --XLReal-- TYPE = ARRAY [0..3] OF CARDINAL;
Black: --Display-- PROCEDURE [
   window: Handle, box: Window.Box, bounds: Window.BoxHandle ^NIL];
BlackParallelogram: --Display-- PROCEDURE [
   window: Handle, p: Parallelogram, dstFunc: DstFunc ^null,
   bounds: Window.BoxHandle ^NIL];
Blanks: --XFormat-- PROCEDURE [h: Handle ^NIL, n: CARDINAL ^1];
Block: --XFormat-- PROCEDURE [h: Handle ^NIL, block: Environment.Block];
Block: --XString-- PROCEDURE [r: Reader]
   RETURNS [block: Environment.Block, context: Context];
BodyEnumProc: --StarWindowShell-- TYPE = PROCEDURE [victim: Window.Handle]
   RETURNS [stop: BOOLEAN ^FALSE];
Boolean: --XToken-- PROCEDURE [h: Handle, signalOnError: BOOLEAN ^TRUE]
   RETURNS [true: BOOLEAN];
BooleanChangeProc: --FormWindow-- TYPE = PROCEDURE [
   window: Window.Handle, item: ItemKey, calledBecauseOf: ChangeReason,
   newValue: BOOLEAN];
BooleanFalseDefault: --PropertySheet-- TYPE = BOOLEAN ^FALSE;
BooleanItemLabel: --FormWindow-- TYPE = RECORD [
   var: SELECT type: BooleanItemLabelType FROM
      string => [string: XString.ReaderBody],
      bitmap => [bitmap: Bitmap],
      ENDCASE];
BooleanItemLabelType: --FormWindow-- TYPE = {string, bitmap};
Box: --KeyboardWindow-- TYPE = RECORD [
   place: Window.Place, width: INTEGER, height: INTEGER];
Box: --Window-- TYPE = RECORD [place: Place, dims: Dims];
BoxEnumProc: --Window-- TYPE = PROCEDURE [Handle, Box];
BoxesAreDisjoint: --Window-- PROCEDURE [a: Box, b: Box] RETURNS [BOOLEAN];
boxFlags: --Display-- BitBltFlags;
BoxHandle: --Window-- TYPE = LONG POINTER TO Box;
Brackets: --XToken-- QuoteProcType;
BreakCharOption: --XString-- TYPE = {ignore, appendToFront, leaveOnRest};
BreakTable: --XString-- TYPE = LONG POINTER TO BreakTableObject;
```

BreakTableObject: --*XString*-- TYPE = RECORD [
  otherSets: StopOrNot ^stop,
  set: Environment.Byte ^0,
  codes: PACKED ARRAY [0..255] OF StopOrNot ^ALL[not]];
Brick: --*Display*-- TYPE = LONG DESCRIPTOR FOR ARRAY CARDINAL OF CARDINAL;
BufferProc: --*SimpleTextDisplay*-- TYPE = PROCEDURE [
  result: Result, string: XString.Reader, address: Environment.BitAddress,
  dims: Window.Dims, bitsPerLine: CARDINAL] RETURNS [continue: BOOLEAN];
Byte: --*XString*-- TYPE = Environment.Byte;
ByteLength: --*XString*-- PROCEDURE [r: Reader] RETURNS [CARDINAL];
Bytes: --*XString*-- TYPE = LONG POINTER TO ByteSequence;
ByteSequence: --*XString*-- TYPE = RECORD [
  PACKED SEQUENCE COMPUTED CARDINAL OF Byte];
CacheFillStatus: --*ContainerCache*-- TYPE = {
  no, inProgress, inProgressPendingAbort, inProgressPendingJoin, yes,
  yesWithError, spare};
CallBack: --*TIP*-- PROCEDURE [
  window: Window.Handle, table: Table, notify: CallBackNotifyProc];
CallBackNotifyProc: --*TIP*-- TYPE = PROCEDURE [
  window: Window.Handle, results: Results] RETURNS [done: BOOLEAN];
CancelPeriodicNotify: --*TIP*-- PROCEDURE [PeriodicNotify]
  RETURNS [null: PeriodicNotify];
CanYouConvert: --*Selection*-- PROCEDURE [
  target: Target, enumeration: BOOLEAN ^FALSE] RETURNS [yes: BOOLEAN];
CanYouTake: --*ContainerSource*-- CanYouTakeProc;
CanYouTakeProc: --*ContainerSource*-- TYPE = PROCEDURE [
  source: Handle, selection: Selection.ConvertProc ^NIL]
  RETURNS [yes: BOOLEAN];
caretRate: --*TIP*-- Process.Ticks;
CatalogProc: --*Catalog*-- TYPE = PROCEDURE [catalogType: NSFile.Type]
  RETURNS [continue: BOOLEAN ^TRUE];
ChangeInfo: --*ContainerSource*-- TYPE = RECORD [
  var: SELECT changeType: ChangeType FROM
    replace = > [item: ItemIndex],
    insert = > [insertInfo: LONG DESCRIPTOR FOR ARRAY CARDINAL OF EditInfo],
    delete = > [deleteInfo: EditInfo],
    all = > NULL,
    noChanges = > NULL,
    ENDCASE];
ChangeProc: --*Containee*-- TYPE = PROCEDURE [
  changeProcData: LONG POINTER, data: DataHandle ^NIL,
  changedAttributes: NSFile.Selections ^[xxxx], noChanges: BOOLEAN ^FALSE];
ChangeProc: --*ContainerSource*-- TYPE = PROCEDURE [
  changeProcData: LONG POINTER, changeInfo: ChangeInfo];
ChangeReason: --*FormWindow*-- TYPE = {user, client, restore};
ChangeScope: --*FileContainerSource*-- PROCEDURE [
  source: ContainerSource.Handle, newScope: NSFile.Scope];
ChangeSizeProc: --*SimpleTextEdit*-- TYPE = PROCEDURE [
  f: Field, oldHeight: INTEGER, newHeight: INTEGER, repaint: BOOLEAN];
ChangeType: --*ContainerSource*-- TYPE = {
  replace, insert, delete, all, noChanges};
Char: --*XFormat*-- PROCEDURE [h: Handle ^NIL, char: XString.Character];

```
Character: --XChar-- TYPE = WORD;
Character: --XString-- TYPE = XChar.Character;
CharacterLength: --XString-- PROCEDURE [r: Reader] RETURNS [CARDINAL];
CharRep: --XChar-- TYPE = MACHINE DEPENDENT RECORD [
   set(0:0..7): Environment.Byte, code(0:8..15): Environment.Byte];
CharTranslator: --TIP-- TYPE = RECORD [proc: KeyToCharProc, data: LONG POINTER];
ChoiceChangeProc: --FormWindow-- TYPE = PROCEDURE [
   window: Window.Handle, item: ItemKey, calledBecauseOf: ChangeReason,
   oldValue: ChoiceIndex, newValue: ChoiceIndex];
ChoiceHintsProc: --FormWindow-- TYPE = PROCEDURE [
   window: Window.Handle, item: ItemKey]
   RETURNS [
      hints: LONG DESCRIPTOR FOR ARRAY CARDINAL OF ChoiceIndex,
      freeHints: FreeChoiceHintsProc];
ChoiceIndex: --FormWindow-- TYPE = CARDINAL [0..37777B];
ChoiceItem: --FormWindow-- TYPE = RECORD [
   var: SELECT type: ChoiceItemType FROM
      string => [choiceNumber: ChoiceIndex, string: XString.ReaderBody],
      bitmap => [choiceNumber: ChoiceIndex, bitmap: Bitmap],
      wrapIndicator => NULL,
      ENDCASE];
ChoiceItems: --FormWindow-- TYPE = LONG DESCRIPTOR FOR ARRAY ChoiceIndex OF
   ChoiceItem;
ChoiceItemType: --FormWindow-- TYPE = {string, bitmap, wrapIndicator};
Circle: --Display-- PROCEDURE [
   window: Handle, place: Window.Place, radius: INTEGER,
   lineStyle: LineStyle ←NIL, bounds: Window.BoxHandle ←NIL];
Clarity: --Window-- TYPE = {isClear, isDirty};
Clear: --Attention-- PROCEDURE;
Clear: --MessageWindow-- PROCEDURE [window: Window.Handle];
Clear: --Selection-- PROCEDURE [unmark: BOOLEAN ←TRUE];
ClearInputFocusOnMatch: --TIP-- PROCEDURE [Window.Handle];
ClearManager: --TIP-- PROCEDURE;
ClearOnMatch: --Selection-- PROCEDURE [
   pointer: ManagerData, unmark: BOOLEAN ←TRUE];
ClearSticky: --Attention-- PROCEDURE;
ClearWriter: --XString-- PROCEDURE [w: Writer];
clickTimeout: --TIP-- System.Pulses;
ClientData: --XFormat-- TYPE = LONG POINTER;
ClientData: --XMessage-- TYPE = LONG POINTER;
clientDirectoryWords: --BWSAttributeTypes-- NSFile.ExtendedAttributeType =
   10373B;
clientFileWords: --BWSAttributeTypes-- NSFile.ExtendedAttributeType = 10372B;
Clients: --ContainerCache-- PROCEDURE [cache: Handle]
   RETURNS [clients: LONG POINTER];
clientSize: --BWSAttributeTypes-- NSFile.ExtendedAttributeType = 10375B;
clientStatus: --BWSAttributeTypes-- NSFile.ExtendedAttributeType = 10374B;
Code: --XChar-- PROCEDURE [c: Character] RETURNS [code: Environment.Byte];
Codes0: --XCharSet0-- TYPE = MACHINE DEPENDENT{
   null, tab(9), lineFeed, formFeed(12), newLine, esc(27), space(32),
   exclamationPoint, neutralDoubleQuote, numberSign, currency, percentSign,
   ampersand, apostrophe, openParenthesis, closeParenthesis, asterisk, plus,
    comma, minus, period, slash, digit0, digit1, digit2, digit3, digit4, digit5,
    digit6, digit7, digit8, digit9, colon, semicolon, lessThan, equals,
    greaterThan, questionMark, commercialAt, upperA, upperB, upperC, upperD,
    upperE, upperF, upperG, upperH, upperI, upperJ, upperK, upperL, upperM,
    upperN, upperO, upperP, upperQ, upperR, upperS, upperT, upperU, upperV,
    upperW, upperX, upperY, upperZ, openBracket, backSlash, closeBracket,
    circumflex, lowBar, grave, lowerA, lowerB, lowerC, lowerD, lowerE, lowerF,
    lowerG, lowerH, lowerI, lowerJ, lowerK, lowerL, lowerM, lowerN, lowerO,
    lowerP, lowerQ, lowerR, lowerS, lowerT, lowerU, lowerV, lowerW, lowerX,
    lowerY, lowerZ, openBrace, verticalBar, closeBrace, tilde,
    invertedExclamation(161), cent, poundSterling, dollar, yen, section(167),
    leftSingleQuote(169), leftDoubleQuote, leftDoubleGuillemet, leftArrow,
    upArrow, rightArrow, downArrow, degree, plusOrMinus, superscript2,
    superscript3, multiply, micro, paragraph, centeredDot, divide,
    rightSingleQuote, rightDoubleQuote, rightDoubleGuillemet, oneQuarter, oneHalf,
    threeQuarters, invertedQuestionMark, graveAccent(193), acuteAccent,
    circumflexAccent, tildeAccent, macronAccent, breveAccent, overDotAccent,
    dieresisAccent, overRingAccent(202), cedilla, underline, doubleAcuteAccent,
    ogonek, hachekAccent, horizontalBar, superscript1, registered, copyright,
    trademark, musicNote, oneEighth(220), threeEighths, fiveEighths, sevenEighths,
    ohmSign, upperAEdigraph, upperDstroke, feminineSpanishOrdinal, upperHstroke,
    upperIJdiagraph(230), upperLdot, upperLstroke, upperOslash, upperOEdiagraph,
    masculineSpanishOrdinal, upperThorn, upperTstroke, upperEng, lowerNapostrophe,
    lowerKgreenlandic, lowerAEdigraph, lowerDstroke, lowerEth, lowerHstroke,
    lowerIdotless, lowerIJdiagraph, lowerLdot, lowerLstroke, lowerOslash,
    lowerOEdiagraph, lowerSzed, lowerThorn, lowerTstroke, lowerEng, escape};
Codes164: --XCharSet164-- TYPE = MACHINE DEPENDENT{
    kabu(33), maruA, maruI, maruU, maruE, maruO, maruRo, maruHa, maruNi, maruHo,
    maruHe, maruTo, maruTi, maruRi, maruNu, reserved(255)};
Codes356: --XCharSet356-- TYPE = MACHINE DEPENDENT{
    thickSpace(33), fourEmSpace, hairSpace, punctuationSpace, decimalPoint(46),
    absoluteValue(124), similarTo(126), escape(255)};
Codes357: --XCharSet357-- TYPE = MACHINE DEPENDENT{
    nonBreakingSpace(33), nonBreakingHyphen, discretionaryHyphen, enDash, emDash,
    figureDash, neutralQuote, loweredLeftDoubleQuote, germanRightDoubleQuote,
    guillemetLeftQuote, guillemetRightQuote, enQuad, emQuad, figureSpace,
    thinSpace, dagger, doubleDagger, bra, ket, rightPointingIndex,
    leftPointingIndex, leftPerp, rightPerp, keft2Perp, right2Perp,
    leftWhiteLenticularBracket, rightWhiteLenticularBracket, nwArrow, seArrow,
    neArrow, swArrow, careOf, perThousand, muchLessThan, muchGreaterThan,
    notLessThan, notGreaterThan, divides, doesNotDivide, parallel, notParallel,
    isAMemberOf, isNotAMemberOf, suchThat, doubleBackArrow, doubleDoubleArrow,
    doubleRightArrow, reversibleReaction2, reversibleReaction1, doubleArrow,
    curlyArrow, contains1, containedIn1, intersection, union, containsOrEquals,
    containedInOrEquals, contains2, containedIn2, neitherConatainsNorIsEqualTo,
    neitherContainedInNorIsEqualTo, doesNotContain, isNotContainedIn,
    checketBallotBox, nullSet, abstractPlus, abstractMinus, abstractTimes,
    abstractDivide, centeredBullet, centeredRing, plancksConstant, litre, not,
    borkenVerticalBar, angle, sphericalAngle, identifier, because, perpendicular,
    isProportionalTo, equivalent, equalByDefinition, questionedEquality, integral,
    contourIntegral, approximatelyEqual1, isomorphic, approximatelyEqual2,
    summation, product, root, minusOrPlus, shade, cruzeiro(161), florin, francs,
    pesetas, europeanCurrency, milreis, genericInfinity, number, take, tel, yogh,
    complexNumber, naturalNumber, realNumber, integer, leftCeiling, rightCeiling,
    leftFloor, rightFloor, therExists, forAll, and, or, qed, nabla,
    partialDerivative, ocrHook, ocrFork, ocrChair, alternatingCurrent,
    doubleLowBar, arc, romanNumeralI, romanNumeralII, romanNumeralIII,
    romanNumeralIV, romanNumeralV, romanNumeralVI, romanNumeralVII,
    romanNumeralVIII, romanNumeralIX, romanNumeralX, spades, hearts, diamonds,
    clubs, checkMark, xMark, circled1, circled2, circled3, circled4, circled5,
    circled6, circled7, circled8, circled9, circled10, circledRightArrow,
    circledRightThenDownArrow, circledDownThenLeftArrow, peaceSymbol, smileFace,
    poison, thickVerticalLine, thickHorizontalLine, thickIntersectingLines,
    thinVerticalLine, thinHorizontalLine, thinIntersectingLines, sun,
    firstQuarterMoon, thirdQuarterMood, mercury, jupiter, saturn, uranus, neptune,
    pluto, aquarius, pisces, aries, taurus, gemini, cancer, leo, virgo, libra,
    scorpius, sagittarius, capricorn, telephone, oneThird, twoThirds, escape};
Codes360: --XCharSet360-- TYPE = MACHINE DEPENDENT{
    ligatureFF(33), ligatureFFI, ligatureFFL, ligatureFI, ligatureFL, ligatureFT,
    sigmaFinal(126), verticalTabGraphic(184), tabGraphic, lineFeedGraphic,
    formFeedGraphic, carriageReturnGraphic, newLineGraphic, available276B,
    available277B, available300B, available301B, pageFormatGraphic,
    startOfDocumentGraphic, stopGraphic, available305B, available306B,
    available307B, available310B, available311B, blackRectGraphic,
    checkerBoardGraphic, ibmDup, available315B, ibmFm, paraTabGraphic(217),
    available332B, available333B, available334B, newParagraphGraphic,
    available336B, available337B, available340B, boxMT, boxNOT, boxEllipsis,
    boxRange, boxUpperX, boxUpperA, boxdigit9, boxUpperZ, boxAsterisk,
    available352B, available353B, boxPlus, boxMinus, boxPeriod, boxComma,
    fieldFormatGreek(246), fieldFormatRussian, fieldFormatHiragana,
    fieldFormatKatakana, fieldFormatKanji, fieldFormatJapanese, spaceGraphicdot,
    spaceGraphicb, escape(255)};
Codes361: --XCharSet361-- TYPE = MACHINE DEPENDENT{
    upperAgrave(33), upperAacute, upperAcircumflex, upperAtilde, upperAmacron,
    upperAbrev, upperAumlaut, upperAring, upperAogonek, upperCacute,
    upperCcircumflex, upperChighDot, upperCcedilla, upperChachek, upperDhachek,
    upperEgrave, upperEacute, upperEcircumflex, upperEmacron, upperEhighDot,
    upperEumlaut, upperEogonek, upperEhachek, upperGcircumflex(57), upperGbrev,
    upperGhighDot, upperGcedilla, upperHcircumflex, upperIgrave, upperIacute,
    upperIcircumflex, upperItilde, upperImacron, upperIhighDot, upperIumlaut,
    upperIogonek, upperJcircumflex, upperKcedilla, upperLacute, upperLcedilla,
    upperLhachek, upperNacute, upperNtilde, upperNcedilla, upperNhachek,
    upperOgrave, upperOacute, upperOcircumflex, upperOtilde, upperOmacron,
    upperOumlaut, upperODoubleAcute, upperRacute, upperRogonek, upperRhachek,
    upperSacute, upperScircumflex, upperScedilla, upperShachek, upperTcedilla,
    upperThachek, upperUgrave, upperUacute, upperUcircumflex, upperUtilde,
    upperUmacron, upperUbrev, upperUumlaut, upperUring, upperUDoubleAcute,
    upperUogonek, upperWcircumflex, upperYgrave, upperYacute, upperYcircumflex,
    upperYumlaut, upperZacute, upperZhighDot, upperZhachek, lowerAgrave(161),
    lowerAacute, lowerAcircumflex, lowerAtilde, lowerAmacron, lowerAbrev,
    lowerAumlaut, lowerAring, lowerAogonek, lowerCacute, lowerCcircumflex,
    lowerChighDot, lowerCcedilla, lowerChachek, lowerDhachek, lowerEgrave,
    lowerEacute, lowerEcircumflex, lowerEmacron, lowerEhighDot, lowerEumlaut,
    lowerEogonek, lowerEhachek, lowerGacute, lowerGcircumflex, lowerGbrev,
    lowerGhighDot, lowerHcircumflex(189), lowerIgrave, lowerIacute,
    lowerIcircumflex, lowerItilde, lowerImacron, lowerIumlaut(196), lowerIogonek,
    lowerJcircumflex, lowerKcedilla, lowerLacute, lowerLcedilla, lowerLhachek,
    lowerNacute, lowerNtilde, lowerNcedilla, lowerNhachek, lowerOgrave,
    lowerOacute, lowerOcircumflex, lowerOtilde, lowerOmacron, lowerOumlaut,
    lowerODoubleAcute, lowerRacute, lowerRogonek, lowerRhachek, lowerSacute,
    lowerScircumflex, lowerScedilla, lowerShachek, lowerTcedilla, lowerThachek,
    lowerUgrave, lowerUacute, lowerUcircumflex, lowerUtilde, lowerUmacron,
    lowerUbrev, lowerUumlaut, lowerUring, lowerUDoubleAcute, lowerUogonek,
    lowerWcircumflex, lowerYgrave, lowerYacute, lowerYcircumflex, lowerYumlaut,
    lowerZacute, lowerZhighDot, lowerZhachek, escape(255)};
Codes41: --XCharSet41-- TYPE = MACHINE DEPENDENT{
    kanjiSpace(33), japaneseComma, japanesePeriod, dakuonMark(43), handakuonMark,
    repeatHiragana(51), repeatHiraganaWithDakuon, repeatKatakana,
    repeatKatakanaWithDakuon, reduplicate, reduplicateAboveItem, repeatKanji,
    shime, kanjiZero, longVowelBar, hyphen(62), parallelSign(66),
    threeDotLeader(68), twoDotLeader, leftBrokenBracket(76), rightBrokenBracket,
    leftJapaneseQuote(86), rightJapaneseQuote, leftJapaneseDoubleQuote,
    rightJapaneseDoubleQuote, leftBlackLenticularBracket,
    rightBlackLenticularBracket, notEqual(98), lessThanOrEqualTo(101),
    greaterThanOrEqualTo, infinity, therefore, male, female, minutes(108),
    seconds, degreesCelsius, whiteStar(121), blackStar, whiteCircle, blackCircle,
    bullsEye, whiteDiamond, escape(255)};
Codes42: --XCharSet42-- TYPE = MACHINE DEPENDENT{
    blackDiamond(33), whiteSquare, blackSquare, whiteUpTriangle, blackUpTriangle,
    whiteDownTriangle, blackDownTriangle, jisKome, jisPostOffice, escape(255)};
Codes43: --XCharSet43-- TYPE = MACHINE DEPENDENT{
    musicalFlat(172), soundRecordingCopyright(174), ayn(176), alifHamzah,
    lowerLeftQuote, musicalSharp(188), mjagkijZnak, tverdyjZnak, risingTone(192),
    umlaut(201), highCommaOffCentre(203), highInvertedComma, horn(206),
    hookToTheLeft(210), circleBelow(212), halfCircleBelow, dotBelow,
    doubleDotBelow, doubleUnderline(217), africanVerticalBar, circumflexUndermark,
    leftHalfOfLigature(221), rightHalfOfLigature, rightHalfOfDoubleTilda,
    escape(255)};
Codes44: --XCharSet44-- TYPE = MACHINE DEPENDENT{
    hirSmallA(33), hirA, hirSmallI, hirI, hirSmallU, hirU, hirSmallE, hirE,
    hirSmallO, hirO, hirKa, hirGa, hirKi, hirGi, hirKu, hirGu, hirKe, hirGe,
    hirKo, hirGo, hirSa, hirZa, hirSi, hirJi, hirSu, hirZu, hirSe, hirZe, hirSo,
    hirZo, hirTa, hirDa, hirTi, hirDi, hirSmallTu, hirTu, hirDu, hirTe, hirDe,
    hirTo, hirDo, hirNa, hirNi, hirNu, hirNe, hirNo, hirHa, hirBa, hirPa, hirHi,
    hirBi, hirPi, hirHu, hirBu, hirPu, hirHe, hirBe, hirPe, hirHo, hirBo, hirPo,
    hirMa, hirMi, hirMu, hirMe, hirMo, hirSmallYa, hirYa, hirSmallYu, hirYu,
    hirSmallYo, hirYo, hirRa, hirRi, hirRu, hirRe, hirRo, hirSmallWa, hirWa,
    hirWi, hirWe, hirWo, hirN, escape(255)};
Codes45: --XCharSet45-- TYPE = MACHINE DEPENDENT{
    katSmallA(33), katA, katSmallI, katI, katSmallU, katU, katSmallE, katE,
    katSmallO, katO, katKa, katGa, katKi, katGi, katKu, katGu, katKe, katGe,
    katKo, katGo, katSa, katZa, katSi, katJi, katSu, katZu, katSe, katZe, katSo,
    katZo, katTa, katDa, katTi, katDi, katSmallTu, katTu, katDu, katTe, katDe,
    katTo, katDo, katNa, katNi, katNu, katNe, katNo, katHa, katBa, katPa, katHi,
    katBi, katPi, katHu, katBu, katPu, katHe, katBe, katPe, katHo, katBo, katPo,
    katMa, katMi, katMu, katMe, katMo, katSmallYa, katYa, katSmallYu, katYu,
   katSmallYo, katYo, katRa, katRi, katRu, katRe, katRo, katSmallWa, katWa,
   katWi, katWe, katWo, katN, katVu, katSmallKa, katSmallKe, escape(255)};
Codes46: --XCharSet46-- TYPE = MACHINE DEPENDENT{
   smootheBreathing(37), roughBreathing, iotaScript, upperPrime(52), lowerPrime,
   raisedPeriod(59), upperAlpha(65), upperBeta, upperGamma(68), upperDelta,
   upperEpsilon, upperStigma, upperDigamma, upperZeta, upperEta, upperTheta,
   upperIota, upperKappa, upperLambda, upperMu, upperNu, upperXi, upperOmicron,
   upperPi, upperKoppa, upperRho, upperSigma, a127B, upperTau, upperUpsilon,
   upperPhi, upperKhi, upperPsi, upperOmega, upperSampi, lowerAlpha(97),
   lowerBeta, lowerBetaMiddleWord, lowerGamma, lowerDelta, lowerEpsilon,
   lowerStigma, lowerDigamma, lowerZeta, lowerEta, lowerTheta, lowerIota,
   lowerKappa, lowerLambda, lowerMu, lowerNu, lowerXi, lowerOmicron, lowerPi,
   lowerKoppa, lowerRho, lowerSigma, lowerSigmaMiddleWord, lowerTau,
   lowerUpsilon, lowerPhi, lowerKhi, lowerPsi, lowerOmega, lowerSampi,
   escape(255)};
Codes47: --XCharSet47-- TYPE = MACHINE DEPENDENT{
   upperA(33), upperBe, upperVe, upperGe, upperDe, upperYe, upperYo, upperZhe,
   upperZe, upperI, upperIKratkoye, upperKa, upperEl, upperEm, upperEn, upperO,
   upperPe, upperEr, upperEs, upperTe, upperU, upperEf, upperXa, upperTse,
   upperChe, upperSha, upperShCha, upperTvyordiiZnak, upperYeri,
   upperMyaxkiiZnak, upperEOborotnoye, upperYu, upperYa, lowerA(81), lowerBe,
   lowerVe, lowerGe, lowerDe, lowerYe, lowerYo, lowerZhe, lowerZe, lowerI,
   lowerIKratkoye, lowerKa, lowerEl, lowerEm, lowerEn, lowerO, lowerPe, lowerEr,
   lowerEs, lowerTe, lowerU, lowerEf, lowerXa, lowerTse, lowerChe, lowerSha,
   lowerShCha, lowerTvyordiiZnak, lowerYeri, lowerMyaxkiiZnak, lowerEOborotnoye,
   lowerYu, lowerYa, escape(255)};
ColumnContents: --FileContainerSource-- TYPE = LONG DESCRIPTOR FOR ARRAY
   CARDINAL OF ColumnContentsInfo;
ColumnContentsInfo: --FileContainerSource-- TYPE = RECORD [
   info: SELECT type: ColumnType FROM
      attribute => [
         attr: NSFile.AttributeType,
         formatProc: AttributeFormatProc ←NIL,
         needsDataHandle: BOOLEAN ←FALSE],
      extendedAttribute => [
         extendedAttr: NSFile.ExtendedAttributeType,
         formatProc: AttributeFormatProc ←NIL,
         needsDataHandle: BOOLEAN ←FALSE],
      multipleAttributes => [
         attrs: NSFile.Selections,
         formatProc: MultiAttributeFormatProc ←NIL,
         needsDataHandle: BOOLEAN ←FALSE],
      ENDCASE];
ColumnCount: --ContainerSource-- ColumnCountProc;
ColumnCountProc: --ContainerSource-- TYPE = PROCEDURE [source: Handle]
   RETURNS [columns: CARDINAL];
ColumnHeaderInfo: --ContainerWindow-- TYPE = RECORD [
   width: CARDINAL, wrap: BOOLEAN ←TRUE, heading: XString.ReaderBody];
ColumnHeaders: --ContainerWindow-- TYPE = LONG DESCRIPTOR FOR ARRAY CARDINAL OF
   ColumnHeaderInfo;
ColumnType: --FileContainerSource-- TYPE = {
   attribute, extendedAttribute, multipleAttributes};
```

```
CommandProc: --FormWindow-- TYPE = PROCEDURE [
   window: Window.Handle, item: ItemKey, clientData: LONG POINTER];
Compare: --XLReal-- PROCEDURE [a: Number, b: Number] RETURNS [Comparison];
Compare: --XString-- PROCEDURE [
   r1: Reader, r2: Reader, ignoreCase: BOOLEAN ^TRUE,
   sortOrder: SortOrder ^standard] RETURNS [Relation];
CompareStringsAndStems: --XString-- PROCEDURE [
   r1: Reader, r2: Reader, ignoreCase: BOOLEAN ^TRUE,
   sortOrder: SortOrder ^standard]
   RETURNS [relation: Relation, equalStems: BOOLEAN];
Comparison: --XLReal-- TYPE = {less, equal, greater};
compatibility: --BWSAttributeTypes-- NSFile.ExtendedAttributeType = 10376B;
Compose: --XMessage-- PROCEDURE [
   source: XString.Reader, destination: XString.Writer, args: StringArray];
ComposeOne: --XMessage-- PROCEDURE [
   source: XString.Reader, destination: XString.Writer, arg: XString.Reader];
ComposeOneToFormatHandle: --XMessage-- PROCEDURE [
   source: XString.Reader, destination: XFormat.Handle, arg: XString.Reader];
ComposeToFormatHandle: --XMessage-- PROCEDURE [
   source: XString.Reader, destination: XFormat.Handle, args: StringArray];
ComputeEndContext: --XString-- PROCEDURE [r: Reader] RETURNS [c: Context];
ConfirmChoices: --Attention-- TYPE = RECORD [
   yes: XString.Reader, no: XString.Reader];
Conic: --Display-- PROCEDURE [
   window: Handle, a: LONG INTEGER, b: LONG INTEGER, c: LONG INTEGER,
   d: LONG INTEGER, e: LONG INTEGER, errorTerm: LONG INTEGER,
   start: Window.Place, stop: Window.Place, errorRef: Window.Place,
   sharpCornered: BOOLEAN, unboundedStart: BOOLEAN, unboundedStop: BOOLEAN,
   lineStyle: LineStyle ^NIL, bounds: Window.BoxHandle ^NIL];
containedIn: --BWSAttributeTypes-- NSFile.ExtendedAttributeType = 10400B;
Context: --XString-- TYPE = MACHINE DEPENDENT RECORD [
   suffixSize(0:0..6): [1..2],
   homogeneous(0:7..7): BOOLEAN,
   prefix(0:8..15): Environment.Byte];
ConversionInfo: --Selection-- TYPE = RECORD [
   SELECT type: * FROM
   convert = > NULL,
   enumeration = > [proc: PROCEDURE [Value] RETURNS [stop: BOOLEAN]],
   query = > [query: LONG DESCRIPTOR FOR ARRAY CARDINAL OF QueryElement],
   ENDCASE];

Dummy: DEFINITIONS =
BEGIN

Convert: --Selection-- PROCEDURE [
   target: Target, zone: UNCOUNTED ZONE ^LOOPHOLE[0]] RETURNS [value: Value];
Converter: --ProductFactoringProducts-- Product = 7;
ConvertInteger: --UnitConversion-- PROCEDURE [
   n: LONG INTEGER, inputUnits: Units, outputUnits: Units]
   RETURNS [LONG INTEGER];
ConvertItem: --ContainerSource-- ConvertItemProc;
ConvertItemProc: --ContainerSource-- TYPE = PROCEDURE [
   source: Handle, itemIndex: ItemIndex, n: CARDINAL ^1,
   target: Selection.Target, zone: UNCOUNTED ZONE,
   info: Selection.ConversionInfo ˆxxx, changeProc: ChangeProc ˆNIL,
   changeProcData: LONG POINTER ˆNIL] RETURNS [value: Selection.Value];
ConvertNumber: --Selection-- PROCEDURE [target: Target]
   RETURNS [ok: BOOLEAN, number: LONG UNSPECIFIED];
ConvertProc: --Selection-- TYPE = PROCEDURE [
   data: ManagerData, target: Target, zone: UNCOUNTED ZONE,
   info: ConversionInfo ˆxxx] RETURNS [value: Value];
ConvertReal: --UnitConversion-- PROCEDURE [
   n: XLReal.Number, inputUnits: Units, outputUnits: Units]
   RETURNS [XLReal.Number];
Copy: --Selection-- PROCEDURE [v: ValueHandle, data: LONG POINTER];
CopyMove: --Selection-- ValueCopyMoveProc;
CopyOrMove: --Selection-- TYPE = {copy, move};
CopyReader: --XString-- PROCEDURE [r: Reader, z: UNCOUNTED ZONE]
   RETURNS [new: Reader];
CopyToNewReaderBody: --XString-- PROCEDURE [r: Reader, z: UNCOUNTED
ZONE]
   RETURNS [ReaderBody];
CopyToNewWriterBody: --XString-- PROCEDURE [
   r: Reader, z: UNCOUNTED ZONE, endContext: Context ˆunknownContext,
   extra: CARDINAL ˆ0] RETURNS [w: WriterBody];
Cos: --XLReal-- PROCEDURE [radians: Number] RETURNS [cos: Number];
coversheetOn: --BWSAttributeTypes-- NSFile.ExtendedAttributeType =
10412B;
CR: --XFormat-- PROCEDURE [h: Handle ˆNIL, n: CARDINAL ˆ1];
Create: --Catalog-- PROCEDURE [
   name: XString.Reader, catalogType: NSFile.Type,
   session: NSFile.Session ˆLOOPHOLE[0]] RETURNS [catalog:
NSFile.Reference];
Create: --ContainerWindow-- PROCEDURE [
   window: Window.Handle, source: ContainerSource.Handle,
   columnHeaders: ColumnHeaders, firstItem: ContainerSource.ItemIndex ˆ0]
   RETURNS [
      regularMenuItems: MenuData.ArrayHandle,
      topPusheeMenuItems: MenuData.ArrayHandle];
Create: --ContainerWindowExtra-- PROCEDURE [
   window: Window.Handle, source: ContainerSource.Handle,
   columnHeaders: ContainerWindow.ColumnHeaders,
   firstItem: ContainerSource.ItemIndex ˆ0, readOnly: BOOLEAN ˆFALSE]
   RETURNS [
      regularMenuItems: MenuData.ArrayHandle,
      topPusheeMenuItems: MenuData.ArrayHandle];
Create: --Context-- PROCEDURE [
   type: Type, data: Data, proc: DestroyProcType, window: Window.Handle];
Create: --FileContainerShell-- PROCEDURE [
   file: NSFile.Reference, columnHeaders: ContainerWindow.ColumnHeaders,
   columnContents: FileContainerSource.ColumnContents,
   regularMenuItems: MenuData.ArrayHandle ˆxxx,
   topPusheeMenuItems: MenuData.ArrayHandle ˆxxx, scope: NSFile.Scope ˆ
   xxx,
   position: ContainerSource.ItemIndex ˆ0,
   options: FileContainerSource.Options ˆLOOPHOLE[0]]
   RETURNS [shell: StarWindowShell.Handle];
```

Create: --*FileContainerSource*-- PROCEDURE [
  file: NSFile.Reference, columns: ColumnContents, scope: NSFile.Scope ^xxx,
  options: Options ^LOOPHOLE[0]] RETURNS [source: ContainerSource.Handle];
Create: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, makeItemsProc: MakeItemsProc,
  layoutProc: LayoutProc ^NIL, windowChangeProc: GlobalChangeProc ^NIL,
  minDimsChangeProc: MinDimsChangeProc ^NIL,
  zone: UNCOUNTED ZONE ^LOOPHOLE[0], clientData: LONG POINTER ^NIL];
Create: --*MessageWindow*-- PROCEDURE [
  window: Window.Handle, zone: UNCOUNTED ZONE ^LOOPHOLE[0],
  lines: CARDINAL ^10];
Create: --*PropertySheet*-- PROCEDURE [
  formWindowItems: FormWindow.MakeItemsProc, menuItemProc: MenuItemProc,
  size: Window.Dims, menuItems: MenuItems ^propertySheetDefaultMenu,
  title: XString.Reader ^NIL, placeToDisplay: Window.Place ^nullPlace,
  formWindowItemsLayout: FormWindow.LayoutProc ^NIL,
  windowAttachedTo: StarWindowShell.Handle ^LOOPHOLE[0],
  globalChangeProc: FormWindow.GlobalChangeProc ^NIL, display: BOOLEAN ^TRUE,
  clientData: LONG POINTER ^NIL, afterTakenDownProc: MenuItemProc ^NIL,
  zone: UNCOUNTED ZONE ^LOOPHOLE[0]] RETURNS [shell: StarWindowShell.Handle];
Create: --*Prototype*-- PROCEDURE [
  name: XString.Reader, type: NSFile.Type, version: Version,
  subtype: Subtype ^0, size: LONG CARDINAL ^0, isDirectory: BOOLEAN ^FALSE,
  session: NSFile.Session ^LOOPHOLE[0]] RETURNS [prototype: NSFile.Handle];
Create: --*StarWindowShell*-- PROCEDURE [
  transitionProc: TransitionProc ^NIL, name: XString.Reader ^NIL,
  namePicture: XString.Character ^0, host: Handle ^LOOPHOLE[0],
  type: ShellType ^regular, sleeps: BOOLEAN ^FALSE,
  considerShowingCoverSheet: BOOLEAN ^TRUE,
  currentlyShowingCoverSheet: BOOLEAN ^FALSE,
  pushersAreReadonly: BOOLEAN ^FALSE, readonly: BOOLEAN ^FALSE,
  scrollData: ScrollData ^vanillaScrollData,
  garbageCollectBodiesProc: PROCEDURE [Handle] ^NIL,
  isCloseLegalProc: IsCloseLegalProc ^NIL, bodyGravity: Window.Gravity ^nw,
  zone: UNCOUNTED ZONE ^LOOPHOLE[0]] RETURNS [Handle];
Create: --*Window*-- PROCEDURE [
  display: DisplayProc, box: Box, parent: Handle ^rootWindow,
  sibling: Handle ^NIL, child: Handle ^NIL, clearingRequired: BOOLEAN ^TRUE,
  windowPane: BOOLEAN ^FALSE, under: BOOLEAN ^FALSE, cookie: BOOLEAN ^FALSE,
  color: BOOLEAN ^FALSE, zone: UNCOUNTED ZONE ^LOOPHOLE[0]]
  RETURNS [window: Handle];
CreateBody: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, repaintProc: PROCEDURE [Window.Handle] ^NIL,
  bodyNotifyProc: TIP.NotifyProc ^NIL, box: Window.Box ^xxx]
  RETURNS [Window.Handle];
CreateCharTable: --*TIP*-- PROCEDURE [
  z: UNCOUNTED ZONE ^LOOPHOLE[0], buffered: BOOLEAN ^TRUE]
  RETURNS [table: Table];
CreateDesktop: --*StarDesktop*-- PROCEDURE [name: XString.Reader]
  RETURNS [fh: NSFile.Handle];
CreateField: --*SimpleTextEdit*-- PROCEDURE [
  clientData: LONG POINTER, context: FieldContext, dims: Window.Dims,
  initString: XString.Reader ^NIL,
  flushness: SimpleTextDisplay.Flushness ^fromFirstChar,
  streakSuccession: SimpleTextDisplay.StreakSuccession ^fromFirstChar,
  readOnly: BOOLEAN ^FALSE, password: BOOLEAN ^FALSE,

fixedHeight: BOOLEAN ^FALSE, font: SimpleTextFont.MappedFontHandle ^NIL,
backingWriter: XString.Writer ^NIL,
SPECIALKeyboard: BlackKeys.Keyboard ^NIL] RETURNS [f: Field];
CreateFieldContext: --*SimpleTextEdit*-- PROCEDURE [
z: UNCOUNTED ZONE, window: Window.Handle, changeSizeProc: ChangeSizeProc]
RETURNS [fc: FieldContext];
CreateFile: --*Catalog*-- PROCEDURE [
catalogType: NSFile.Type ^10476B, name: XString.Reader, type: NSFile.Type,
isDirectory: BOOLEAN ^FALSE, size: LONG CARDINAL ^0,
session: NSFile.Session ^LOOPHOLE[0]] RETURNS [file: NSFile.Handle];
CreateItem: --*MenuData*-- PROCEDURE [
zone: UNCOUNTED ZONE, name: XString.Reader, proc: MenuProc,
itemData: LONG UNSPECIFIED ^0] RETURNS [ItemHandle];
CreateLinked: --*PropertySheet*-- PROCEDURE [
formWindowItems: FormWindow.MakeItemsProc, menuItemProc: MenuItemProc,
size: Window.Dims, menuItems: MenuItems ^propertySheetDefaultMenu,
title: XString.Reader ^NIL, placeToDisplay: Window.Place ^nullPlace,
formWindowItemsLayout: FormWindow.LayoutProc ^NIL,
windowAttachedTo: StarWindowShell.Handle ^LOOPHOLE[0],
globalChangeProc: FormWindow.GlobalChangeProc ^NIL, display: BOOLEAN ^TRUE,
linkWindowItems: FormWindow.MakeItemsProc,
linkWindowItemsLayout: FormWindow.LayoutProc ^NIL,
clientData: LONG POINTER ^NIL, afterTakenDownProc: MenuItemProc ^NIL,
zone: UNCOUNTED ZONE ^LOOPHOLE[0]] RETURNS [shell: StarWindowShell.Handle];
CreateMenu: --*MenuData*-- PROCEDURE [
zone: UNCOUNTED ZONE, title: ItemHandle, array: ArrayHandle,
copyItemsIntoMenusZone: BOOLEAN ^FALSE] RETURNS [MenuHandle];
CreatePeriodicNotify: --*TIP*-- PROCEDURE [
window: Window.Handle ^NIL, results: Results, milliSeconds: CARDINAL,
notifyProc: NotifyProc ^NIL] RETURNS [PeriodicNotify];
CreatePlaceHolderTable: --*TIP*-- PROCEDURE [z: UNCOUNTED ZONE ^LOOPHOLE[0]]
RETURNS [table: Table];
CreateProcType: --*Context*-- TYPE = PROCEDURE RETURNS [Data, DestroyProcType];
CreateTable: --*TIP*-- PROCEDURE [
file: XString.Reader, z: UNCOUNTED ZONE ^LOOPHOLE[0],
contents: XString.Reader ^NIL] RETURNS [table: Table];
Current: --*XTime*-- PROCEDURE RETURNS [time: System.GreenwichMeanTime];
DashCnt: --*Display*-- CARDINAL = 6;
Data: --*Containee*-- TYPE = RECORD [reference: NSFile.Reference ^xxx];
Data: --*Context*-- TYPE = LONG POINTER;
DataHandle: --*Containee*-- TYPE = LONG POINTER TO Data;
Date: --*XFormat*-- PROCEDURE [
h: Handle ^NIL, time: System.GreenwichMeanTime ^LOOPHOLE[17601311200B],
format: DateFormat ^dateAndTime];
dateAndTime: --*XTime*-- XString.Reader;
DateColumn: --*FileContainerSource*-- PROCEDURE
RETURNS [multipleAttributes ColumnContentsInfo];
DateFormat: --*XFormat*-- TYPE = {dateOnly, timeOnly, dateAndTime};
dateOnly: --*XTime*-- XString.Reader;
DaysOfWeek: --*XComSoftMessage*-- TYPE = Keys [monday..sunday];
Decase: --*XChar*-- PROCEDURE [c: Character] RETURNS [Character];
Decimal: --*XFormat*-- PROCEDURE [h: Handle ^NIL, n: LONG INTEGER];
Decimal: --*XToken*-- PROCEDURE [h: Handle, signalOnError: BOOLEAN ^TRUE]
RETURNS [i: LONG INTEGER];
DecimalFormat: --*XFormat*-- NumberFormat;

Decompose: --*XMessage*-- PROCEDURE [source: XString.Reader]
  RETURNS [args: StringArray];
DefaultFileConvertProc: --*Containee*-- Selection.ConvertProc;
defaultGeometry: --*KeyboardWindow*-- BlackKeys.GeometryTable;
DefaultLayout: --*FormWindow*-- LayoutProc;
defaultPicture: --*KeyboardWindow*-- BlackKeys.Picture;
DefaultPictureProc: --*KeyboardWindow*-- BlackKeys.PictureProc;
defaultTabStops: --*FormWindow*-- TabStops;
defaultTime: --*XTime*-- System.GreenwichMeanTime;
Defined: --*Cursor*-- TYPE = Type [blank..column];
DeleteAll: --*Undo*-- PROCEDURE;
DeleteAndShowNextPrevious: --*ContainerWindow*-- PROCEDURE [
  window: Window.Handle, item: ContainerSource.ItemIndex, direction: Direction];
DeleteAndShowNextPrevious: --*ContainerWindowExtra2*-- PROCEDURE [
  window: Window.Handle, item: ContainerSource.ItemIndex,
  direction: ContainerWindow.Direction]
  RETURNS [newOpenShell: StarWindowShell.Handle];
DeleteItems: --*ContainerSource*-- DeleteItemsProc;
DeleteItemsProc: --*ContainerSource*-- TYPE = PROCEDURE [
  source: Handle, itemIndex: ItemIndex, n: CARDINAL ^1,
  changeProc: ChangeProc ^NIL, changeProcData: LONG POINTER ^NIL];
DeleteNItems: --*ContainerCache*-- PROCEDURE [
  cache: Handle, item: CARDINAL, nitems: CARDINAL ^1];
Delimited: --*XToken*-- FilterProcType;
Dependency: --*Event*-- TYPE [2];
Dereference: --*XString*-- PROCEDURE [r: Reader] RETURNS [rb: ReaderBody];
DescribeOption: --*ProductFactoring*-- PROCEDURE [
  option: Option, desc: XString.Reader,
  prerequisite: Prerequisite ^nullPrerequisite];
DescribeProduct: --*ProductFactoring*-- PROCEDURE [
  product: Product, desc: XString.Reader];
DescribeReader: --*XString*-- Courier.Description;
DescribeReaderBody: --*XString*-- Courier.Description;
desktop: --*BWSFileTypes*-- NSFile.Type = 10400B;
desktopCatalog: --*BWSFileTypes*-- NSFile.Type = 10400B;
DesktopProc: --*IdleControl*-- TYPE = PROCEDURE;
desktopWindowAvailable: --*StarDesktop*-- Atom.ATOM;
DestallBody: --*StarWindowShell*-- PROCEDURE [body: Window.Handle];
Destroy: --*ContainerWindow*-- PROCEDURE [Window.Handle];
Destroy: --*Context*-- PROCEDURE [type: Type, window: Window.Handle];
Destroy: --*FormWindow*-- PROCEDURE [window: Window.Handle];
Destroy: --*MessageWindow*-- PROCEDURE [Window.Handle];
Destroy: --*StarWindowShell*-- PROCEDURE [sws: Handle];
DestroyAll: --*Context*-- PROCEDURE [window: Window.Handle];
DestroyBody: --*StarWindowShell*-- PROCEDURE [body: Window.Handle];
DestroyField: --*SimpleTextEdit*-- PROCEDURE [f: Field];
DestroyFieldContext: --*SimpleTextEdit*-- PROCEDURE [fc: FieldContext];
END.

DestroyItem: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, repaint: BOOLEAN ^TRUE];
DestroyItem: --*MenuData*-- PROCEDURE [zone: UNCOUNTED ZONE, item: ItemHandle];
DestroyItems: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: LONG DESCRIPTOR FOR ARRAY CARDINAL OF ItemKey,
  repaint: BOOLEAN ^TRUE];

DestroyMenu: --*MenuData*-- PROCEDURE [zone: UNCOUNTED ZONE, menu: MenuHandle];
DestroyMessages: --*XMessage*-- PROCEDURE [h: Handle];
DestroyMsgsProc: --*XMessage*-- TYPE = PROCEDURE [clientData: ClientData];
DestroyProcType: --*Context*-- TYPE = PROCEDURE [Data, Window.Handle];
DestroyTable: --*TIP*-- PROCEDURE [LONG POINTER TO Table];
DFonts: --*ProductFactoringProducts*-- Product = 3;
Difficulty: --*Selection*-- TYPE = {easy, moderate, hard, impossible};
Digit: --*XLReal*-- TYPE = [0..9];
Digits: --*XLReal*-- TYPE = PACKED ARRAY [0..12] OF Digit;
Dims: --*Window*-- TYPE = RECORD [w: INTEGER, h: INTEGER];
Direction: --*ContainerWindow*-- TYPE = {next, previous};
Discard: --*Selection*-- PROCEDURE [saved: Saved, unmark: BOOLEAN ^TRUE];
DisplayProc: --*Window*-- TYPE = PROCEDURE [window: Handle];
Divide: --*XLReal*-- PROCEDURE [a: Number, b: Number] RETURNS [Number];
DoAnUndo: --*Undo*-- PROCEDURE;
DoAnUnundo: --*Undo*-- PROCEDURE;
DoneLookingAtTextItemValue: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey];
DoneWithString: --*AtomicProfile*-- PROCEDURE [string: XString.Reader];
dontTimeout: --*Attention*-- Process.Ticks = 0;
DoTheGreeterProc: --*IdleControl*-- GreeterProc;
Double: --*XLReal*-- PROCEDURE [Number] RETURNS [Number];
DownUp: --*LevelIVKeys*-- TYPE = KeyStations.DownUp;
DownUp: --*TIP*-- TYPE = LevelIVKeys.DownUp;
DstFunc: --*Display*-- TYPE = BitBlt.DstFunc;
E: --*XLReal*-- PROCEDURE RETURNS [Number];
EditInfo: --*ContainerSource*-- TYPE = RECORD [
  afterItem: ItemIndex, nItems: CARDINAL];
Ellipse: --*Display*-- PROCEDURE [
  window: Handle, center: Window.Place, xRadius: INTEGER, yRadius: INTEGER,
  lineStyle: LineStyle ^NIL, bounds: Window.BoxHandle ^NIL];
Empty: --*XString*-- PROCEDURE [r: Reader] RETURNS [BOOLEAN];
emptyContext: --*XString*-- Context;
Enabled: --*ProductFactoring*-- PROCEDURE [option: Option]
  RETURNS [enabled: BOOLEAN];
EntireBox: --*Window*-- PROCEDURE [Handle] RETURNS [box: Box];
EntryEnumProc: --*OptionFile*-- TYPE = PROCEDURE [entry: XString.Reader]
  RETURNS [stop: BOOLEAN ^FALSE];
Enumerate: --*Catalog*-- PROCEDURE [proc: CatalogProc];
Enumerate: --*Selection*-- PROCEDURE [
  proc: EnumerationProc, target: Target, data: RequestorData ^NIL,
  zone: UNCOUNTED ZONE ^LOOPHOLE[0]] RETURNS [aborted: BOOLEAN];
EnumerateAllMenus: --*StarWindowShellExtra*-- PROCEDURE [
  sws: StarWindowShell.Handle, proc: StarWindowShell.MenuEnumProc];
EnumerateDisplayed: --*StarWindowShell*-- PROCEDURE [proc: ShellEnumProc]
  RETURNS [Handle ^LOOPHOLE[0]];
EnumerateDisplayedOfType: --*StarWindowShell*-- PROCEDURE [
  type: ShellType, proc: ShellEnumProc] RETURNS [Handle ^LOOPHOLE[0]];
EnumerateEntries: --*OptionFile*-- PROCEDURE [
  section: XString.Reader, callBack: EntryEnumProc,
  file: NSFile.Reference ^xxx];
EnumerateInvalidBoxes: --*Window*-- PROCEDURE [window: Handle, proc: BoxEnumProc];
EnumerateKeyboards: --*KeyboardKey*-- PROCEDURE [
  class: KeyboardClass, enumProc: EnumerateProc];
EnumerateMyDisplayedParasites: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, proc: ShellEnumProc] RETURNS [Handle ^LOOPHOLE[0]];

EnumeratePopupMenus: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, proc: MenuEnumProc];
EnumerateProc: --*KeyboardKey*-- TYPE = PROCEDURE [
  keyboard: BlackKeys.Keyboard, class: KeyboardClass]
  RETURNS [stop: BOOLEAN ^FALSE];
EnumerateSections: --*OptionFile*-- PROCEDURE [
  callBack: SectionEnumProc, file: NSFile.Reference ^xxx];
EnumerateString: --*AtomicProfile*-- PROCEDURE [
  atom: Atom.ATOM, proc: PROCEDURE [XString.Reader]];
EnumerateTree: --*Window*-- PROCEDURE [
  root: Handle, proc: PROCEDURE [window: Handle]];
EnumerationProc: --*Selection*-- TYPE = PROCEDURE [
  element: Value, data: RequestorData] RETURNS [stop: BOOLEAN ^FALSE];
Equal: --*XLReal*-- PROCEDURE [a: Number, b: Number] RETURNS [BOOLEAN];
Equal: --*XString*-- PROCEDURE [r1: Reader, r2: Reader] RETURNS [BOOLEAN];
Equivalent: --*XString*-- PROCEDURE [r1: Reader, r2: Reader] RETURNS [BOOLEAN];
eraseFlags: --*Display*-- BitBltFlags;
Error: --*Containee*-- ERROR [
  msg: XString.Reader ^NIL, error: ERROR ^NIL, errorData: LONG POINTER ^NIL];
Error: --*ContainerSource*-- ERROR [
  code: ErrorCode, msg: XString.Reader ^NIL, error: ERROR ^NIL,
  errorData: LONG POINTER ^NIL];
Error: --*ContainerWindow*-- ERROR [code: ErrorCode];
Error: --*Context*-- ERROR [code: ErrorCode];
Error: --*FormWindow*-- ERROR [code: ErrorCode];
Error: --*KeyboardKey*-- ERROR [code: ErrorCode];
Error: --*OptionFile*-- ERROR [code: ErrorCode];
Error: --*ProductFactoring*--ERROR [type: ErrorType];
Error: --*PropertySheet*-- ERROR [code: ErrorCode];
Error: --*Selection*-- ERROR [code: ErrorCode];
Error: --*SimpleTextEdit*-- ERROR [type: ErrorType];
Error: --*StarWindowShell*-- ERROR [code: ErrorCode];
Error: --*TIP*-- ERROR [code: ErrorCode];
Error: --*Window*-- ERROR [code: ErrorCode];
Error: --*XFormat*-- ERROR [code: ErrorCode];
Error: --*XLReal*-- ERROR [code: ErrorCode];
Error: --*XMessage*-- ERROR [type: ErrorType];
Error: --*XString*-- ERROR [code: ErrorCode];
ErrorCode: --*ContainerSource*-- TYPE = MACHINE DEPENDENT{
  invalidParameters, accessError, fileError, noSuchItem, other, last(15)};
ErrorCode: --*ContainerWindow*-- TYPE = MACHINE DEPENDENT{
  notAContainerWindow, noSuchItem, last(7)};
ErrorCode: --*Context*-- TYPE = {duplicateType, windowIsNIL, tooManyTypes, other};
ErrorCode: --*FormWindow*-- TYPE = MACHINE DEPENDENT{
  notAFormWindow, wrongItemType, invalidChoiceNumber, noSuchLine,
  alreadyAFormWindow, invalidItemKey, itemNotOnLine, duplicateItemKey,
  incompatibleLayout, alreadyLaidOut, last(15)};
ErrorCode: --*KeyboardKey*-- TYPE = {
  alreadyInSystemKeyboards, notInSystemKeyboards, insufficientSpace};
ErrorCode: --*OptionFile*-- TYPE = {
  invalidParameters, inconsistentValue, notFound, syntaxError};
ErrorCode: --*PropertySheet*-- TYPE = {notAPropSheet};
ErrorCode: --*Selection*-- TYPE = {
  tooManyActions, tooManyTargets, invalidOperation, operationFailed, didntAbort,
  didntClear};

ErrorCode: *--StarWindowShell--* TYPE = {
  desktopNotUp, notASWS, notStarStyle, tooManyWindows};
ErrorCode: *--TIP--* TYPE = {noSuchPeriodicNotifier, other};
ErrorCode: *--Window--* TYPE = {
  illegalBitmap, illegalFloat, windowNotChildOfParent, whosSlidingRoot,
  noSuchSibling, noUnderVariant, windowInTree, sizingWithBitmapUnder,
  illegalStack, invalidParameters};
ErrorCode: *--XFormat--* TYPE = {invalidFormat, nilData};
ErrorCode: *--XLReal--* TYPE = {
  bug, divideByZero, invalidOperation, notANumber, overflow, underflow,
  unimplemented};
ErrorCode: *--XString--* TYPE = {
  invalidOperation, multipleCharSets, tooManyBytes, invalidParameter};
ErrorType: *--ProductFactoring--* TYPE = {
  dataNotFound, notStarted, illegalProduct, illegalOption, missingProduct,
  missingOption};
ErrorType: *--SimpleTextEdit--* TYPE = {
  fieldIsNoplace, noRoomInWriter, lastCharGTfirstChar};
ErrorType: *--XMessage--* TYPE = {
  arrayMismatch, invalidMsgKeyList, invalidStringArray, invalidString,
  notEnoughArguments};
EventData: *--ApplicationFolder--* TYPE = RECORD [
  applicationFolder: NSFile.Reference, internalName: XString.Reader];
EventType: *--Event--* TYPE = Atom.ATOM;
Exp: *--XLReal--* PROCEDURE [Number] RETURNS [Number];
ExpandWriter: *--XString--* PROCEDURE [w: Writer, extra: CARDINAL];
Fetch: *--Cursor--* PROCEDURE [h: Handle];
FetchFromType: *--Cursor--* PROCEDURE [h: Handle, type: Defined];
Field: *--SimpleTextEdit--* TYPE = LONG POINTER TO FieldObject;
FieldContext: *--SimpleTextEdit--* TYPE = LONG POINTER TO FieldContextObject;
FieldContextObject: *--SimpleTextEdit--* TYPE;
FieldObject: *--SimpleTextEdit--* TYPE;
fiftyPercent: *--Display--* Brick;
filedrawerReference: *--BWSAttributeTypes--* NSFile.ExtendedAttributeType =
  10407B;
FillProc: *--ContainerCache--* TYPE = PROCEDURE [cache: Handle]
  RETURNS [errored: BOOLEAN ←FALSE];
FillResolveBuffer: *--SimpleTextDisplay--* PROCEDURE [
  string: XString.Reader, lineWidth: CARDINAL ←177777B,
  wordBreak: BOOLEAN ←TRUE, streakSuccession: StreakSuccession ←fromFirstChar,
  resolve: ResolveBuffer, font: SimpleTextFont.MappedFontHandle ←NIL]
  RETURNS [width: CARDINAL, result: Result, rest: XString.ReaderBody];
Filtered: *--XToken--* PROCEDURE [
  h: Handle, data: FilterState, filter: FilterProcType,
  skip: SkipMode ←whiteSpace, temporary: BOOLEAN ←TRUE]
  RETURNS [value: XString.ReaderBody];
FilterProcType: *--XToken--* TYPE = PROCEDURE [
  c: XChar.Character, data: FilterState] RETURNS [inClass: BOOLEAN];
FilterState: *--XToken--* TYPE = LONG POINTER TO StandardFilterState;
Find: *--Context--* PROCEDURE [type: Type, window: Window.Handle] RETURNS [Data];
Find: *--Prototype--* PROCEDURE [
  type: NSFile.Type, version: Version, subtype: Subtype ←0,
  session: NSFile.Session ←LOOPHOLE[0]] RETURNS [reference: NSFile.Reference];
FindDescriptionFile: *--ApplicationFolder--* PROCEDURE [
  applicationFolder: NSFile.Handle] RETURNS [descriptionFile: NSFile.Reference];

FindOrCreate: *--Context--* PROCEDURE [
  type: Type, window: Window.Handle, createProc: CreateProcType] RETURNS [Data];
First: *--XString--* PROCEDURE [r: Reader] RETURNS [c: Character];
firstAvailableApplicationType: *--BWSAttributeTypes--*
  NSFile.ExtendedAttributeType = 10505B;
firstBWSType: *--BWSAttributeTypes--* NSFile.ExtendedAttributeType = 10400B;
firstOldApplicationSpecific: *--BWSAttributeTypes--*
  NSFile.ExtendedAttributeType = 10414B;
firstSpareBWSType: *--BWSAttributeTypes--* NSFile.ExtendedAttributeType = 10461B;
firstStarType: *--BWSFileTypes--* NSFile.Type = 10400B;
Fix: *--XLReal--* PROCEDURE [Number] RETURNS [LONG INTEGER];
FixdPtNum: *--Display--* TYPE = MACHINE DEPENDENT RECORD [
  SELECT OVERLAID * FROM
  wholeThing => [li(0:0..31): LONG INTEGER],
  parts => [frac(0:0..15): CARDINAL, int(1:0..15): INTEGER],
  ENDCASE];
Float: *--Window--* PROCEDURE [window: Handle, temp: Handle, proc: FloatProc];
Float: *--XLReal--* PROCEDURE [LONG INTEGER] RETURNS [Number];
FloatProc: *--Window--* TYPE = PROCEDURE [window: Handle]
  RETURNS [place: Place, done: BOOLEAN];
Flushness: *--FormWindow--* TYPE = SimpleTextDisplay.Flushness;
Flushness: *--SimpleTextDisplay--* TYPE = {flushLeft, flushRight, fromFirstChar};
FlushUserInput: *--TIP--* PROCEDURE;
FocusTakesInput: *--TIP--* PROCEDURE RETURNS [BOOLEAN];
FontNotFound: *--SimpleTextFont--* SIGNAL [name: XString.Reader];
Format: *--XTime--* PROCEDURE [
  xfh: XFormat.Handle ^NIL, time: System.GreenwichMeanTime ^defaultTime,
  template: XString.Reader ^dateAndTime, ltp: LTP ^useSystem];
formatHandle: *--Attention--* XFormat.Handle;
FormatProc: *--XFormat--* TYPE = PROCEDURE [r: XString.Reader, h: Handle];
FormatReal: *--XLReal--* PROCEDURE [
  h: XFormat.Handle ^NIL, r: Number, width: NATURAL];
FractionPart: *--XLReal--* PROCEDURE [Number] RETURNS [Number];
Free: *--Selection--* PROCEDURE [v: ValueHandle];
Free: *--Window--* PROCEDURE [window: Handle, zone: UNCOUNTED ZONE ^LOOPHOLE[0]];
FreeBadPhosphorList: *--Window--* PROCEDURE [window: Handle];
FreeCache: *--ContainerCache--* PROCEDURE [Handle];
FreeChoiceHintsProc: *--FormWindow--* TYPE = PROCEDURE [
  window: Window.Handle, item: ItemKey,
  hints: LONG DESCRIPTOR FOR ARRAY CARDINAL OF ChoiceIndex];
FreeChoiceItems: *--FormWindowMessageParse--* PROCEDURE [
  choiceItems: FormWindow.ChoiceItems, zone: UNCOUNTED ZONE];
FreeContext: *--Selection--* PROCEDURE [v: ValueHandle, zone: UNCOUNTED ZONE];
FreeDataProcedure: *--Event--* TYPE = PROCEDURE [myData: LONG POINTER];
FreeMark: *--ContainerCache--* PROCEDURE [mark: Mark];
FreeMsgDomainsStorage: *--XMessage--* PROCEDURE [msgDomains: MsgDomains];
FreeReaderBytes: *--XString--* PROCEDURE [r: Reader, z: UNCOUNTED ZONE];
FreeReaderHandle: *--XToken--* PROCEDURE [h: Handle] RETURNS [nil: Handle];
FreeResolveBuffer: *--SimpleTextDisplay--* PROCEDURE [ResolveBuffer];
FreeStd: *--Selection--* ValueFreeProc;
FreeStreamHandle: *--XToken--* PROCEDURE [h: Handle] RETURNS [s: Stream.Handle];
FreeTextHintsProc: *--FormWindow--* TYPE = PROCEDURE [
  window: Window.Handle, item: ItemKey,
  hints: LONG DESCRIPTOR FOR ARRAY CARDINAL OF XString.ReaderBody];
FreeTokenString: *--XToken--* PROCEDURE [r: XString.Reader]
  RETURNS [nil: XString.Reader ^NIL];

FreeTree: --*Window*-- PROCEDURE [
  window: Handle, zone: UNCOUNTED ZONE ^LOOPHOLE[0]];
FreeWriterBytes: --*XString*-- PROCEDURE [w: Writer];
FromBlock: --*XString*-- PROCEDURE [
  block: Environment.Block, context: Context ^vanillaContext]
  RETURNS [ReaderBody];
FromChar: --*XString*-- PROCEDURE [char: LONG POINTER TO Character]
  RETURNS [ReaderBody];
FromName: --*ApplicationFolder*-- PROCEDURE [internalName: XString.Reader]
  RETURNS [applicationFolder: NSFile.Reference];
FromNSString: --*XString*-- PROCEDURE [
  s: NSString.String, homogeneous: BOOLEAN ^FALSE] RETURNS [ReaderBody];
FromSTRING: --*XString*-- PROCEDURE [s: LONG STRING, homogeneous: BOOLEAN ^FALSE]
  RETURNS [ReaderBody];
fullUserName: --*StarDesktop*-- Atom.ATOM;
GenericProc: --*Containee*-- TYPE = PROCEDURE [
  atom: Atom.ATOM, data: DataHandle, changeProc: ChangeProc ^NIL,
  changeProcData: LONG POINTER ^NIL] RETURNS [LONG UNSPECIFIED];
GeometryTable: --*BlackKeys*-- TYPE = LONG POINTER;
GeometryTableEntry: --*KeyboardWindow*-- TYPE = RECORD [
  box: Box, key: KeyStations, shift: ShiftState];
Get: --*XMessage*-- PROCEDURE [h: Handle, msgKey: MsgKey]
  RETURNS [msg: XString.ReaderBody];
GetAdjustProc: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [AdjustProc];
GetAvailableBodyWindowDims: --*StarWindowShell*-- PROCEDURE [sws: Handle]
  RETURNS [Window.Dims];
GetBitmapUnder: --*Window*-- PROCEDURE [window: Handle] RETURNS [LONG POINTER];
GetBody: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [Window.Handle];
GetBodyWindowJustFits: --*StarWindowShell*-- PROCEDURE [sws: Handle]
  RETURNS [BOOLEAN];
GetBOOLEAN: --*AtomicProfile*-- PROCEDURE [atom: Atom.ATOM] RETURNS [BOOLEAN];
GetBooleanItemValue: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey] RETURNS [value: BOOLEAN];
GetBooleanValue: --*OptionFile*-- PROCEDURE [
  section: XString.Reader, entry: XString.Reader, file: NSFile.Reference ^xxx]
  RETURNS [value: BOOLEAN];
GetBox: --*SimpleTextEdit*-- PROCEDURE [f: Field] RETURNS [box: Window.Box];
GetBox: --*Window*-- PROCEDURE [Handle] RETURNS [box: Box];
GetCachedName: --*Containee*-- PROCEDURE [data: DataHandle]
  RETURNS [name: XString.ReaderBody, ticket: Ticket];
GetCachedType: --*Containee*-- PROCEDURE [data: DataHandle]
  RETURNS [type: NSFile.Type];
GetCaretPlace: --*SimpleTextEdit*-- PROCEDURE [context: FieldContext]
  RETURNS [place: Window.Place];
GetCharProcType: --*XToken*-- TYPE = PROCEDURE [h: Handle]
  RETURNS [c: XChar.Character];
GetCharTranslator: --*TIP*-- PROCEDURE [table: Table] RETURNS [o: CharTranslator];
GetCharWidth: --*SimpleTextDisplay*-- PROCEDURE [
  char: XChar.Character, font: SimpleTextFont.MappedFontHandle ^NIL]
  RETURNS [width: CARDINAL];
GetChild: --*Window*-- PROCEDURE [Handle] RETURNS [Handle];
GetChoiceItemValue: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey] RETURNS [value: ChoiceIndex];
GetClearingRequired: --*Window*-- PROCEDURE [Handle] RETURNS [BOOLEAN];
GetClientData: --*FormWindow*-- PROCEDURE [window: Window.Handle]
  RETURNS [clientData: LONG POINTER];

GetClientData: --*SimpleTextEdit*-- PROCEDURE [f: Field]
RETURNS [clientData: LONG POINTER];
GetContainee: --*StarWindowShell*-- PROCEDURE [sws: Handle]
RETURNS [Containee.Data];
GetContainerSource: --*FileContainerShell*-- PROCEDURE [
shell: StarWindowShell.Handle] RETURNS [source: ContainerSource.Handle];
GetContainerWindow: --*FileContainerShell*-- PROCEDURE [
shell: StarWindowShell.Handle] RETURNS [window: Window.Handle];
GetCurrentDesktopFile: --*StarDesktop*-- PROCEDURE RETURNS [NSFile.Reference];
GetCurrentKeyboard: --*BlackKeys*-- PROCEDURE RETURNS [current: Keyboard];
GetDecimalItemValue: --*FormWindow*-- PROCEDURE [
window: Window.Handle, item: ItemKey] RETURNS [value: XLReal.Number];
GetDefaultImplementation: --*Containee*-- PROCEDURE RETURNS [Implementation];
GetDesktopProc: --*IdleControl*-- PROCEDURE [atom: Atom.ATOM]
RETURNS [DesktopProc];
GetDims: --*Window*-- PROCEDURE [Handle] RETURNS [dims: Dims];
GetDisplayProc: --*Window*-- PROCEDURE [Handle] RETURNS [DisplayProc];
GetDisplayWindow: --*KeyboardWindow*-- PROCEDURE RETURNS [Window.Handle];
GetFieldContext: --*SimpleTextEdit*-- PROCEDURE [f: Field] RETURNS [FieldContext];
GetFile: --*Catalog*-- PROCEDURE [
catalogType: NSFile.Type ^10476B, name: XString.Reader,
readonly: BOOLEAN ^FALSE, session: NSFile.Session ^LOOPHOLE[0]]
RETURNS [file: NSFile.Handle];
GetFlushness: --*FormWindow*-- PROCEDURE [window: Window.Handle, item: ItemKey]
RETURNS [old: Flushness];
GetFlushness: --*SimpleTextEdit*-- PROCEDURE [f: Field]
RETURNS [SimpleTextDisplay.Flushness];
GetFont: --*SimpleTextEdit*-- PROCEDURE [f: Field]
RETURNS [SimpleTextFont.MappedFontHandle];
GetFormWindows: --*PropertySheet*-- PROCEDURE [shell: StarWindowShell.Handle]
RETURNS [form: Window.Handle, link: Window.Handle];
GetGlobalChangeProc: --*FormWindow*-- PROCEDURE [window: Window.Handle]
RETURNS [proc: GlobalChangeProc];
GetGreeterProc: --*IdleControl*-- PROCEDURE RETURNS [GreeterProc];
GetHandle: --*XComSoftMessage*-- PROCEDURE RETURNS [h: XMessage.Handle];
GetHost: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [Handle];
GetImplementation: --*Containee*-- PROCEDURE [NSFile.Type]
RETURNS [Implementation];
GetImplementation: --*Undo*-- PROCEDURE RETURNS [Implementation];
GetInfo: --*Cursor*-- PROCEDURE RETURNS [info: Info];
GetInputFocus: --*SimpleTextEdit*-- PROCEDURE [fc: FieldContext] RETURNS [Field];
GetInputFocus: --*TIP*-- PROCEDURE RETURNS [Window.Handle];
GetIntegerItemValue: --*FormWindow*-- PROCEDURE [
window: Window.Handle, item: ItemKey] RETURNS [value: LONG INTEGER];
GetIntegerValue: --*OptionFile*-- PROCEDURE [
section: XString.Reader, entry: XString.Reader, index: CARDINAL ^0,
file: NSFile.Reference ^xxx] RETURNS [value: LONG INTEGER];
GetIsCloseLegalProc: --*StarWindowShell*-- PROCEDURE [sws: Handle]
RETURNS [IsCloseLegalProc];
GetItemInfo: --*FileContainerSource*-- PROCEDURE [
source: ContainerSource.Handle, itemIndex: ContainerSource.ItemIndex]
RETURNS [file: NSFile.Reference, type: NSFile.Type];
GetJoinDirection: --*XChar*-- PROCEDURE [Character] RETURNS [JoinDirection];
GetLength: --*ContainerCacheExtra*-- PROCEDURE [cache: ContainerCache.Handle]
RETURNS [cacheLength: CARDINAL];
GetLength: --*ContainerSource*-- GetLengthProc;

GetLengthProc: --*ContainerSource*-- TYPE = PROCEDURE [source: Handle]
  RETURNS [length: CARDINAL, totalOrPartial: TotalOrPartial ^total];
GetLimitProc: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [LimitProc];
GetList: --*XMessage*-- PROCEDURE [
  h: Handle, msgKeys: MsgKeyList, msgs: StringArray];
GetLONGINTEGER: --*AtomicProfile*-- PROCEDURE [atom: Atom.ATOM]
  RETURNS [LONG INTEGER];
GetManager: --*TIP*-- PROCEDURE RETURNS [current: Manager];
GetMode: --*TIPStar*-- PROCEDURE RETURNS [mode: Mode];
GetMultipleChoiceItemValue: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, zone: UNCOUNTED ZONE]
  RETURNS [value: LONG DESCRIPTOR FOR ARRAY CARDINAL OF ChoiceIndex];
GetNextAvailableKey: --*FormWindow*-- PROCEDURE [window: Window.Handle]
  RETURNS [key: ItemKey];
GetNextOutOfProc: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey] RETURNS [NextOutOfProc];
GetNextUnobscuredBox: --*StarDesktop*-- PROCEDURE [height: INTEGER]
  RETURNS [Window.Box];
GetNotifyProc: --*TIP*-- PROCEDURE [window: Window.Handle] RETURNS [NotifyProc];
GetNotifyProcFromTable: --*TIP*-- PROCEDURE [table: Table] RETURNS [NotifyProc];
GetNthItem: --*ContainerCache*-- PROCEDURE [cache: Handle, n: CARDINAL]
  RETURNS [ItemHandle];
GetOpenItem: --*ContainerWindow*-- PROCEDURE [window: Window.Handle]
  RETURNS [item: ContainerSource.ItemIndex ^177777B];
GetPane: --*Window*-- PROCEDURE [Handle] RETURNS [BOOLEAN];
GetParent: --*Window*-- PROCEDURE [Handle] RETURNS [Handle];
GetPlace: --*TIP*-- PROCEDURE [window: Window.Handle] RETURNS [Window.Place];
GetPlaceFromReference: --*StarDesktop*-- PROCEDURE [ref: NSFile.Reference]
  RETURNS [Window.Place];
GetPName: --*Atom*-- PROCEDURE [atom: ATOM] RETURNS [pName: XString.Reader];
GetProp: --*Atom*-- PROCEDURE [onto: ATOM, prop: ATOM] RETURNS [pair: RefPair];
GetPusheeCommands: --*StarWindowShell*-- PROCEDURE [sws: Handle]
  RETURNS [
    bottom: MenuData.MenuHandle, middle: MenuData.MenuHandle,
    top: MenuData.MenuHandle];
GetReadOnly: --*FormWindow*-- PROCEDURE [window: Window.Handle, item: ItemKey]
  RETURNS [readOnly: BOOLEAN];
GetReadOnly: --*SimpleTextEdit*-- PROCEDURE [f: Field]
  RETURNS [readOnly: BOOLEAN];
GetReadonly: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [BOOLEAN];
GetRegularCommands: --*StarWindowShell*-- PROCEDURE [sws: Handle]
    RETURNS [MenuData.MenuHandle];
GetResults: --*TIPX*-- PROCEDURE [
    window: Window.Handle, resultsWanted: ResultsWanted ^NIL]
    RETURNS [results: TIP.Results];
GetScrollData: --*StarWindowShell*-- PROCEDURE [sws: Handle]
    RETURNS [scrollData: ScrollData]

GetSelection: --*ContainerWindow*-- PROCEDURE [window: Window.Handle]
RETURNS [
first: ContainerSource.ItemIndex, lastPlusOne: ContainerSource.ItemIndex];
GetShellFromReference: --*StarDesktop*-- PROCEDURE [ref: NSFile.Reference]
RETURNS [sws: StarWindowShell.Handle];
GetShowKeyboardProc: --*KeyboardKey*-- PROCEDURE RETURNS [ShowKeyboardProc];
GetSibling: --*Window*-- PROCEDURE [Handle] RETURNS [Handle];
GetSleeps: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [BOOLEAN];
GetSource: --*ContainerWindow*-- PROCEDURE [window: Window.Handle]
RETURNS [source: ContainerSource.Handle];
GetState: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [State];
GetStreakNature: --*XChar*-- PROCEDURE [Character] RETURNS [StreakNature];
GetStreakSuccession: --*FormWindow*-- PROCEDURE [
window: Window.Handle, item: ItemKey] RETURNS [old: StreakSuccession];
GetStreakSuccession: --*SimpleTextEdit*-- PROCEDURE [f: Field]
RETURNS [SimpleTextDisplay.StreakSuccession];
GetString: --*AtomicProfile*-- PROCEDURE [atom: Atom.ATOM]
RETURNS [XString.Reader];
GetStringValue: --*OptionFile*-- PROCEDURE [
section: XString.Reader, entry: XString.Reader,
callBack: PROCEDURE [value: XString.Reader], index: CARDINAL ^0,
file: NSFile.Reference ^xxx];
GetTable: --*TIP*-- PROCEDURE [window: Window.Handle] RETURNS [Table];
GetTable: --*TIPStar*-- PROCEDURE [Placeholder] RETURNS [TIP.Table];
GetTableLink: --*TIP*-- PROCEDURE [from: Table] RETURNS [to: Table];
GetTableOpacity: --*TIP*-- PROCEDURE [table: Table] RETURNS [BOOLEAN];
GetTabStops: --*FormWindow*-- PROCEDURE [window: Window.Handle]
RETURNS [tabStops: TabStops];
GetTag: --*FormWindow*-- PROCEDURE [window: Window.Handle, item: ItemKey]
RETURNS [tag: XString.ReaderBody];
GetTextItemValue: --*FormWindow*-- PROCEDURE [
window: Window.Handle, item: ItemKey, zone: UNCOUNTED ZONE]
RETURNS [value: XString.ReaderBody];
GetTransitionProc: --*StarWindowShell*-- PROCEDURE [sws: Handle]
RETURNS [TransitionProc];
GetType: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [ShellType];
GetUseBadPhosphor: --*Window*-- PROCEDURE [Handle] RETURNS [BOOLEAN];
GetUserProfile: --*OptionFile*-- PROCEDURE RETURNS [file: NSFile.Reference];
GetValue: --*SimpleTextEdit*-- PROCEDURE [f: Field] RETURNS [XString.ReaderBody];
GetVisibility: --*FormWindow*-- PROCEDURE [window: Window.Handle, item: ItemKey]
RETURNS [visibility: Visibility];
GetWindow: --*SimpleTextEdit*-- PROCEDURE [fc: FieldContext]
RETURNS [window: Window.Handle];
GetWindow: --*StarDesktop*-- PROCEDURE RETURNS [Window.Handle];
GetWindowItemValue: --*FormWindow*-- PROCEDURE [
window: Window.Handle, item: ItemKey] RETURNS [value: Window.Handle];
GetWorkstationProfile: --*OptionFile*-- PROCEDURE
RETURNS [file: NSFile.Reference];
GetZone: --*FormWindow*-- PROCEDURE [window: Window.Handle]
RETURNS [zone: UNCOUNTED ZONE];
GetZone: --*SimpleTextEdit*-- PROCEDURE [fc: FieldContext]
RETURNS [UNCOUNTED ZONE];
GetZone: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [UNCOUNTED ZONE];

GlobalChangeProc: --*FormWindow*-- TYPE = PROCEDURE [
  window: Window.Handle, item: ItemKey, calledBecauseOf: ChangeReason,
  clientData: LONG POINTER];
Gravity: --*Window*-- TYPE = {nil, nw, n, ne, e, se, s, sw, w, c, xxx};
Gray: --*Display*-- PROCEDURE [
  window: Handle, box: Window.Box, gray: Brick ^fiftyPercent,
  dstFunc: DstFunc ^null, bounds: Window.BoxHandle ^NIL];
GrayTrapezoid: --*Display*-- PROCEDURE [
  window: Handle, t: Trapezoid, gray: Brick ^fiftyPercent,
  dstFunc: DstFunc ^null, bounds: Window.BoxHandle ^NIL];
Greater: --*XLReal*-- PROCEDURE [a: Number, b: Number] RETURNS [BOOLEAN];
GreaterEq: --*XLReal*-- PROCEDURE [a: Number, b: Number] RETURNS [BOOLEAN];
GreeterProc: --*IdleControl*-- TYPE = PROCEDURE RETURNS [Atom.ATOM];
Half: --*XLReal*-- PROCEDURE [Number] RETURNS [Number];
Handle: --*ContainerCache*-- TYPE = LONG POINTER TO Object;
Handle: --*ContainerSource*-- TYPE = LONG POINTER TO Procedures;
Handle: --*Cursor*-- TYPE = LONG POINTER TO Object;
Handle: --*Display*-- TYPE = Window.Handle;
Handle: --*StarWindowShell*-- TYPE = RECORD [Window.Handle];
Handle: --*Window*-- TYPE = LONG POINTER TO Object;
Handle: --*XFormat*-- TYPE = LONG POINTER TO Object;
Handle: --*XMessage*-- TYPE = LONG POINTER TO Object;
Handle: --*XToken*-- TYPE = LONG POINTER TO Object;
HasAnyBeenChanged: --*FormWindow*-- PROCEDURE [window: Window.Handle]
  RETURNS [yes: BOOLEAN];
HasBeenChanged: --*FormWindow*-- PROCEDURE [window: Window.Handle, item: ItemKey]
  RETURNS [yes: BOOLEAN];
HaveDisplayedParasite: --*StarWindowShell*-- PROCEDURE [sws: Handle]
  RETURNS [BOOLEAN];
Hex: --*XFormat*-- PROCEDURE [h: Handle ^NIL, n: LONG CARDINAL];
HexFormat: --*XFormat*-- NumberFormat;
HighlightThisKey: --*SoftKeys*-- PROCEDURE [
  window: Window.Handle, key: CARDINAL ^nullKey];
HostNumber: --*XFormat*-- PROCEDURE [
  h: Handle ^NIL, hostNumber: System.HostNumber, format: NetFormat];
HowHard: --*Selection*-- PROCEDURE [target: Target, enumeration: BOOLEAN ^FALSE]
  RETURNS [difficulty: Difficulty];
IconColumn: --*FileContainerSource*-- PROCEDURE
  RETURNS [attribute ColumnContentsInfo];
Idle: --*IdleControl*-- PROCEDURE;
ignoreType: --*Containee*-- NSFile.Type = 37777777777B;
Implementation: --*Containee*-- TYPE = RECORD [
  implementors: LONG POINTER ^NIL,
  name: XString.ReaderBody ^xxx,
  smallPictureProc: SmallPictureProc ^NIL,
  pictureProc: PictureProc ^NIL,
  convertProc: Selection.ConvertProc ^NIL,
  genericProc: GenericProc ^NIL];
Implementation: --*Undo*-- TYPE = RECORD [
  opportunity: Proc,
  roadblock: PROCEDURE [XString.Reader],
  doAnUndo: PROCEDURE,
  doAnUnundo: PROCEDURE,
  deleteAll: PROCEDURE];

IndexFromMark: --*ContainerCache*-- PROCEDURE [mark: Mark]
  RETURNS [index: CARDINAL];
Info: --*Cursor*-- TYPE = RECORD [type: Type, hotX: [0..15], hotY: [0..15]];
Info: --*FileContainerSource*-- PROCEDURE [source: ContainerSource.Handle]
  RETURNS [
    file: NSFile.Reference, columns: ColumnContents, scope: NSFile.Scope,
    options: Options];
Info: --*SoftKeys*-- PROCEDURE [window: Window.Handle]
  RETURNS [
    table: TIP.Table, notifyProc: TIP.NotifyProc, labels: Labels,
    highlightedKey: CARDINAL, outlinedKey: CARDINAL];
InitBreakTable: --*XString*-- PROCEDURE [
  r: Reader, stopOrNot: StopOrNot, otherSets: StopOrNot]
  RETURNS [break: BreakTableObject];
Initialize: --*Window*-- PROCEDURE [
  window: Handle, display: DisplayProc, box: Box, parent: Handle ^rootWindow,
  sibling: Handle ^NIL, child: Handle ^NIL, clearingRequired: BOOLEAN ^TRUE,
  windowPane: BOOLEAN ^FALSE, under: BOOLEAN ^FALSE, cookie: BOOLEAN ^FALSE,
  color: BOOLEAN ^FALSE];
InitializeWindow: --*Window*-- PROCEDURE [
  window: Handle, display: DisplayProc, box: Box, parent: Handle ^rootWindow,
  sibling: Handle ^NIL, child: Handle ^NIL, clearingRequired: BOOLEAN ^TRUE,
  windowPane: BOOLEAN ^FALSE, under: BOOLEAN ^FALSE, cookie: BOOLEAN ^FALSE,
  color: BOOLEAN ^FALSE];
InsertIntoTree: --*Window*-- PROCEDURE [window: Handle];
InsertItem: --*ContainerCache*-- PROCEDURE [
  cache: Handle, before: CARDINAL, addData: AddData]
  RETURNS [handle: ItemHandle];
InsertItem: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, line: Line, beforeItem: ItemKey,
  preMargin: CARDINAL ^0, tabStop: CARDINAL ^nextTabStop,
  repaint: BOOLEAN ^TRUE];
InsertLine: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, before: Line, spaceAboveLine: CARDINAL ^0]
  RETURNS [line: Line];
InstallBody: --*StarWindowShell*-- PROCEDURE [sws: Handle, body: Window.Handle];
InstallFormWindow: --*PropertySheet*-- PROCEDURE [
  shell: StarWindowShell.Handle, menuItemProc: MenuItemProc,
  menuItems: MenuItems ^propertySheetDefaultMenu, title: XString.Reader ^NIL,
  formWindow: Window.Handle, afterTakenDownProc: MenuItemProc ^NIL];
InsufficientRoom: --*XString*-- SIGNAL [
  needsMoreRoom: Writer, amountNeeded: CARDINAL];
IntegerPart: --*XLReal*-- PROCEDURE [Number] RETURNS [Number];
Interpolator: --*Display*-- TYPE = RECORD [val: FixdPtNum, dVal: FixdPtNum];
IntersectBoxes: --*Window*-- PROCEDURE [b1: Box, b2: Box] RETURNS [box: Box];
Invalid: --*XTime*-- ERROR;
InvalidateBox: --*Window*-- PROCEDURE [
  window: Handle, box: Box, clarity: Clarity ^isDirty];
InvalidateCache: --*Containee*-- PROCEDURE [data: DataHandle];
InvalidateWholeCache: --*Containee*-- PROCEDURE;
InvalidEncoding: --*XString*-- ERROR [
  invalidReader: Reader, firstBadByteOffset: CARDINAL];
InvalidHandle: --*BlackKeys*-- ERROR;
InvalidHandle: --*SoftKeys*-- ERROR;
InvalidNumber: --*XString*-- SIGNAL;
InvalidTable: --*TIP*-- SIGNAL [type: TableError, message: XString.Reader];

Invert: --*Cursor*-- PROCEDURE RETURNS [BOOLEAN];
Invert: --*Display*-- PROCEDURE [
  window: Handle, box: Window.Box, bounds: Window.BoxHandle ^NIL];
IsBitmapUnderVariant: --*Window*-- PROCEDURE [Handle] RETURNS [BOOLEAN];
IsBodyWindowOutOfInterior: --*StarWindowShell*-- PROCEDURE [body: Window.Handle]
  RETURNS [BOOLEAN];
IsCloseLegal: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, closeAll: BOOLEAN ^FALSE] RETURNS [BOOLEAN];
IsCloseLegalProc: --*StarWindowShell*-- TYPE = PROCEDURE [
  sws: Handle, closeAll: BOOLEAN ^FALSE] RETURNS [BOOLEAN];
IsCloseLegalProcReturnsFalse: --*StarWindowShell*-- IsCloseLegalProc;
IsColorVariant: --*Window*-- PROCEDURE [Handle] RETURNS [BOOLEAN];
IsCookieVariant: --*Window*-- PROCEDURE [Handle] RETURNS [BOOLEAN];
IsDescendantOfRoot: --*Window*-- PROCEDURE [Handle] RETURNS [BOOLEAN];
IsIt: --*ContainerWindow*-- PROCEDURE [window: Window.Handle]
  RETURNS [yes: BOOLEAN];
IsIt: --*FileContainerSource*-- PROCEDURE [source: ContainerSource.Handle]
  RETURNS [BOOLEAN];
IsIt: --*FormWindow*-- PROCEDURE [window: Window.Handle] RETURNS [yes: BOOLEAN];
IsIt: --*MessageWindow*-- PROCEDURE [Window.Handle] RETURNS [yes: BOOLEAN];
IsPlaceInBox: --*Window*-- PROCEDURE [place: Place, box: Box] RETURNS [BOOLEAN];
IsSpecial: --*XLReal*-- PROCEDURE [Number]
  RETURNS [yes: BOOLEAN, index: SpecialIndex];
Item: --*MenuData*-- TYPE = PrivateItem;
Item: --*XToken*-- PROCEDURE [h: Handle, temporary: BOOLEAN ^TRUE]
  RETURNS [value: XString.ReaderBody];
ItemClients: --*ContainerCache*-- PROCEDURE [item: ItemHandle]
  RETURNS [clientData: LONG POINTER];
ItemClientsLength: --*ContainerCache*-- PROCEDURE [handle: ItemHandle]
  RETURNS [dataLength: CARDINAL];
ItemData: --*MenuData*-- PROCEDURE [item: ItemHandle] RETURNS [LONG UNSPECIFIED];
ItemGeneric: --*ContainerSource*-- ItemGenericProc;
ItemGenericProc: --*ContainerSource*-- TYPE = PROCEDURE [
  source: Handle, itemIndex: ItemIndex, atom: Atom.ATOM,
  changeProc: ChangeProc ^NIL, changeProcData: LONG POINTER ^NIL]
  RETURNS [LONG UNSPECIFIED];
ItemHandle: --*ContainerCache*-- TYPE = LONG POINTER TO ItemObject;
ItemHandle: --*MenuData*-- TYPE = LONG POINTER TO Item;
ItemIndex: --*ContainerCache*-- PROCEDURE [item: ItemHandle]
  RETURNS [index: CARDINAL];
ItemIndex: --*ContainerSource*-- TYPE = CARDINAL;
ItemKey: --*FormWindow*-- TYPE = CARDINAL;
ItemName: --*MenuData*-- PROCEDURE [item: ItemHandle]
  RETURNS [name: XString.ReaderBody];
ItemNameWidth: --*MenuData*-- PROCEDURE [item: ItemHandle] RETURNS [CARDINAL];
ItemNthString: --*ContainerCache*-- PROCEDURE [item: ItemHandle, n: CARDINAL]
  RETURNS [XString.ReaderBody];
ItemObject: --*ContainerCache*-- TYPE;
ItemProc: --*MenuData*-- PROCEDURE [item: ItemHandle] RETURNS [proc: MenuProc];
ItemStringCount: --*ContainerCache*-- PROCEDURE [item: ItemHandle]
  RETURNS [strings: CARDINAL];
ItemType: --*FormWindow*-- TYPE = MACHINE DEPENDENT{
  choice, multiplechoice, decimal, integer, boolean, text, command, tagonly,
  window, last(15)};
JoinDirection: --*XChar*-- TYPE = {nextCharToRight, nextCharToLeft};

```
KeyBits: --LevelIVKeys-- TYPE = PACKED ARRAY KeyName OF DownUp;
KeyBits: --TIP-- TYPE = LevelIVKeys.KeyBits;
Keyboard: --BlackKeys-- TYPE = LONG POINTER TO KeyboardObject ←NIL;
KeyboardClass: --KeyboardKey-- TYPE = {system, client, special, all, none};
KeyboardObject: --BlackKeys-- TYPE = RECORD [
   table: TIP.Table ←NIL,
   charTranslator: TIP.CharTranslator ←xxx,
   pictureProc: PictureProc ←NIL,
   label: XString.ReaderBody ←xxx,
   clientData: LONG POINTER ←NIL];
KeyName: --LevelIVKeys-- TYPE = MACHINE DEPENDENT{
   notAKey, Keyset1(8), Keyset2, Keyset3, Keyset4, Keyset5, MouseLeft,
   MouseRight, MouseMiddle, Five, Four, Six, E, Seven, D, U, V, Zero, K, Minus,
   P, Slash, Font, Same, BS, Three, Two, W, Q, S, A, Nine, I, X, O, L, Comma,
   CloseQuote, RightBracket, Open, Keyboard, One, Tab, ParaTab, F, Props, C, J,
   B, Z, LeftShift, Period, SemiColon, NewPara, OpenQuote, Delete, Next, R, T, G,
   Y, H, Eight, N, M, Lock, Space, LeftBracket, Equal, RightShift, Stop, Move,
   Undo, Margins, R9, L10, L7, L4, L1, A9, R10, A8, Copy, Find, Again, Help,
   Expand, R4, D2, D1, Center, T1, Bold, Italics, Underline, Superscript,
   Subscript, Smaller, T10, R3, Key47, A10, Defaults, A11, A12};
KeyName: --TIP-- TYPE = LevelIVKeys.KeyName;
Keys: --XComSoftMessage-- TYPE = MACHINE DEPENDENT{
   time, date, dateAndTime, am, pm, january, february, march, april, may, june,
   july, august, september, october, november, december, monday, tuesday,
   wednesday, thursday, friday, saturday, sunday, decimalSeparator,
   thousandsSeparator};
KeyStations: --KeyboardWindow-- TYPE = MACHINE DEPENDENT{
   k1, k2, k3, k4, k5, k6, k7, k8, k9, k10, k11, k12, k13, k14, k15, k16, k17,
   k18, k19, k20, k21, k22, k23, k24, k25, k26, k27, k28, k29, k30, k31, k32,
   k33, k34, k35, k36, k37, k38, k39, k40, k41, k42, k43, k44, k45, k46, k47,
   k48, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, last(96)};
KeyToCharProc: --TIP-- TYPE = PROCEDURE [
   keys: LONG POINTER TO KeyBits, key: KeyName, downUp: DownUp,
   data: LONG POINTER, buffer: XString.Writer];
LabelRecord: --SoftKeys-- TYPE = RECORD [
   unshifted: XString.ReaderBody ←xxx, shifted: XString.ReaderBody ←xxx];
Labels: --SoftKeys-- TYPE = LONG DESCRIPTOR FOR ARRAY CARDINAL OF LabelRecord;
lasstOldApplicationSpecific: --BWSAttributeTypes--
   NSFile.ExtendedAttributeType = 10457B;
lastBWSType: --BWSAttributeTypes-- NSFile.ExtendedAttributeType = 10777B;
LayoutError: --FormWindow-- SIGNAL [code: LayoutErrorCode];
LayoutErrorCode: --FormWindow-- TYPE = {onTopOfAnotherItem, notEnufTabsDefined};
LayoutInfoFromItem: --FormWindow-- PROCEDURE [
   window: Window.Handle, item: ItemKey]
   RETURNS [line: Line, margin: CARDINAL, tabStop: CARDINAL, box: Window.Box];
LayoutProc: --FormWindow-- TYPE = PROCEDURE [
   window: Window.Handle, clientData: LONG POINTER];
Less: --XLReal-- PROCEDURE [a: Number, b: Number] RETURNS [BOOLEAN];
LessEq: --XLReal-- PROCEDURE [a: Number, b: Number] RETURNS [BOOLEAN];
LimitProc: --StarWindowShell-- TYPE = PROCEDURE [sws: Handle, box: Window.Box]
   RETURNS [Window.Box];
Line: --Display-- PROCEDURE [
   window: Handle, start: Window.Place, stop: Window.Place,
   lineStyle: LineStyle ←NIL, bounds: Window.BoxHandle ←NIL];
Line: --FormWindow-- TYPE [2];
```

```
Line: --XFormat-- PROCEDURE [
   h: Handle ←NIL, r: XString.Reader, n: CARDINAL ←1];
Line: --XToken-- FilterProcType;
LineStyle: --Display-- TYPE = LONG POINTER TO LineStyleObject;
LineStyleObject: --Display-- TYPE = RECORD [
   widths: ARRAY [0..5] OF CARDINAL, thickness: CARDINAL];
LineUpBoxes: --FormWindow-- PROCEDURE [
   window: Window.Handle,
   items: LONG DESCRIPTOR FOR ARRAY CARDINAL OF ItemKey ←xxx];
Ln: --XLReal-- PROCEDURE [Number] RETURNS [Number];
Log: --XLReal-- PROCEDURE [base: Number, arg: Number] RETURNS [Number];
logoff: --StarDesktop-- Atom.ATOM;
logon: --StarDesktop-- Atom.ATOM;
LogonSession: --BWSZone-- PROCEDURE RETURNS [UNCOUNTED ZONE];
logonSession: --BWSZone-- UNCOUNTED ZONE;
LookAtTextItemValue: --FormWindow-- PROCEDURE [
   window: Window.Handle, item: ItemKey] RETURNS [value: XString.ReaderBody];
Lop: --XString-- PROCEDURE [r: Reader] RETURNS [c: Character];
LosingFocusProc: --TIP-- TYPE = PROCEDURE [
   w: Window.Handle, data: LONG POINTER];
LowerCase: --XChar-- PROCEDURE [c: Character] RETURNS [Character];
LTP: --XTime-- TYPE = RECORD [
   r: SELECT t: * FROM
      useSystem = > NULL, useThese = > [ltp: System.LocalTimeParameters], ENDCASE];
mailStatus: --BWSAttributeTypes-- NSFile.ExtendedAttributeType = 10411B;
Make: --Atom-- PROCEDURE [pName: XString.Reader] RETURNS [atom: ATOM];
Make: --XChar-- PROCEDURE [set: Environment.Byte, code: Environment.Byte]
   RETURNS [Character];
Make: --XCharSet0-- PROCEDURE [code: Codes0] RETURNS [XChar.Character];
Make: --XCharSet164-- PROCEDURE [code: Codes164] RETURNS [XChar.Character];
Make: --XCharSet356-- PROCEDURE [code: Codes356] RETURNS [XChar.Character];
Make: --XCharSet357-- PROCEDURE [code: Codes357] RETURNS [XChar.Character];
Make: --XCharSet360-- PROCEDURE [code: Codes360] RETURNS [XChar.Character];
Make: --XCharSet361-- PROCEDURE [code: Codes361] RETURNS [XChar.Character];
Make: --XCharSet41-- PROCEDURE [code: Codes41] RETURNS [XChar.Character];
Make: --XCharSet42-- PROCEDURE [code: Codes42] RETURNS [XChar.Character];
Make: --XCharSet43-- PROCEDURE [code: Codes43] RETURNS [XChar.Character];
Make: --XCharSet44-- PROCEDURE [code: Codes44] RETURNS [XChar.Character];
Make: --XCharSet45-- PROCEDURE [code: Codes45] RETURNS [XChar.Character];
Make: --XCharSet46-- PROCEDURE [code: Codes46] RETURNS [XChar.Character];
Make: --XCharSet47-- PROCEDURE [code: Codes47] RETURNS [XChar.Character];
MakeAtom: --Atom-- PROCEDURE [pName: LONG STRING] RETURNS [atom: ATOM];
MakeBooleanItem: --FormWindow-- PROCEDURE [
   window: Window.Handle, myKey: ItemKey, tag: XString.Reader ←NIL,
   suffix: XString.Reader ←NIL, visibility: Visibility ←visible,
   boxed: BOOLEAN ←TRUE, readOnly: BOOLEAN ←FALSE,
   changeProc: BooleanChangeProc ←NIL, label: BooleanItemLabel,
   initBoolean: BOOLEAN ←TRUE];
MakeChoiceItem: --FormWindow-- PROCEDURE [
   window: Window.Handle, myKey: ItemKey, tag: XString.Reader ←NIL,
   suffix: XString.Reader ←NIL, visibility: Visibility ←visible,
   boxed: BOOLEAN ←TRUE, readOnly: BOOLEAN ←FALSE, values: ChoiceItems,
   initChoice: ChoiceIndex, fullyDisplayed: BOOLEAN ←TRUE,
   verticallyDisplayed: BOOLEAN ←FALSE, hintsProc: ChoiceHintsProc ←NIL,
   changeProc: ChoiceChangeProc ←NIL,
   outlineOrHighlight: OutlineOrHighlight ←highlight];
```

MakeCommandItem: --*FormWindow*-- PROCEDURE [
window: Window.Handle, myKey: ItemKey, tag: XString.Reader ^NIL,
suffix: XString.Reader ^NIL, visibility: Visibility ^visible,
boxed: BOOLEAN ^TRUE, readOnly: BOOLEAN ^FALSE, commandProc: CommandProc,
commandName: XString.Reader, clientData: LONG POINTER ^NIL];
MakeDecimalItem: --*FormWindow*-- PROCEDURE [
window: Window.Handle, myKey: ItemKey, tag: XString.Reader ^NIL,
suffix: XString.Reader ^NIL, visibility: Visibility ^visible,
boxed: BOOLEAN ^TRUE, readOnly: BOOLEAN ^FALSE, signed: BOOLEAN ^FALSE,
width: CARDINAL, initDecimal: XLReal.Number ^xxx,
wrapUnderTag: BOOLEAN ^FALSE, hintsProc: TextHintsProc ^NIL,
nextOutOfProc: NextOutOfProc ^NIL, displayTemplate: XString.Reader ^NIL,
SPECIALKeyboard: BlackKeys.Keyboard ^NIL];
MakeIntegerItem: --*FormWindow*-- PROCEDURE [
window: Window.Handle, myKey: ItemKey, tag: XString.Reader ^NIL,
suffix: XString.Reader ^NIL, visibility: Visibility ^visible,
boxed: BOOLEAN ^TRUE, readOnly: BOOLEAN ^FALSE, signed: BOOLEAN ^FALSE,
width: CARDINAL, initInteger: LONG INTEGER ^0, wrapUnderTag: BOOLEAN ^FALSE,
hintsProc: TextHintsProc ^NIL, nextOutOfProc: NextOutOfProc ^NIL,
SPECIALKeyboard: BlackKeys.Keyboard ^NIL];
MakeItemsProc: --*FormWindow*-- TYPE = PROCEDURE [
window: Window.Handle, clientData: LONG POINTER];
MakeMenuItem: --*FormWindow*-- PROCEDURE [
window: Window.Handle, myKey: ItemKey, tag: XString.Reader ^NIL,
suffix: XString.Reader ^NIL, visibility: Visibility ^visible,
boxed: BOOLEAN ^TRUE, menu: MenuData.MenuHandle];
MakeMultipleChoiceItem: --*FormWindow*-- PROCEDURE [
window: Window.Handle, myKey: ItemKey, tag: XString.Reader ^NIL,
suffix: XString.Reader ^NIL, visibility: Visibility ^visible,
boxed: BOOLEAN ^TRUE, readOnly: BOOLEAN ^FALSE, values: ChoiceItems,
initChoice: LONG DESCRIPTOR FOR ARRAY CARDINAL OF ChoiceIndex,
fullyDisplayed: BOOLEAN ^TRUE, verticallyDisplayed: BOOLEAN ^FALSE,
hintsProc: ChoiceHintsProc ^NIL, changeProc: MultipleChoiceChangeProc ^NIL,
outlineOrHighlight: OutlineOrHighlight ^highlight];
MakeNegative: --*Cursor*-- PROCEDURE;
MakePositive: --*Cursor*-- PROCEDURE;
MakeSpecial: --*XLReal*-- PROCEDURE [index: SpecialIndex] RETURNS [Number];
MakeTagOnlyItem: --*FormWindow*-- PROCEDURE [
window: Window.Handle, myKey: ItemKey, tag: XString.Reader,
visibility: Visibility ^visible];
MakeTextItem: --*FormWindow*-- PROCEDURE [
window: Window.Handle, myKey: ItemKey, tag: XString.Reader ^NIL,
suffix: XString.Reader ^NIL, visibility: Visibility ^visible,
boxed: BOOLEAN ^TRUE, readOnly: BOOLEAN ^FALSE, width: CARDINAL,
initString: XString.Reader ^NIL, wrapUnderTag: BOOLEAN ^FALSE,
passwordFeedback: BOOLEAN ^FALSE, hintsProc: TextHintsProc ^NIL,
nextOutOfProc: NextOutOfProc ^NIL,
SPECIALKeyboard: BlackKeys.Keyboard ^NIL];
MakeWindowItem: --*FormWindow*-- PROCEDURE [
window: Window.Handle, myKey: ItemKey, tag: XString.Reader ^NIL,
visibility: Visibility ^visible, boxed: BOOLEAN ^TRUE, size: Window.Dims,
nextIntoProc: NextIntoProc ^NIL] RETURNS [clientWindow: Window.Handle];
Manager: --*TIP*-- TYPE = RECORD [
table: Table, window: Window.Handle, notify: NotifyProc];
ManagerData: --*Selection*-- TYPE = LONG POINTER;

Map: --*XString*-- PROCEDURE [r: Reader, proc: MapCharProc]
RETURNS [c: Character];
MapAtomProc: --*Atom*-- TYPE = PROCEDURE [ATOM] RETURNS [BOOLEAN];
MapAtoms: --*Atom*-- PROCEDURE [proc: MapAtomProc] RETURNS [lastAtom: ATOM];
MapCharProc: --*XString*-- TYPE = PROCEDURE [c: Character]
RETURNS [stop: BOOLEAN];
MappedDefaultFont: --*SimpleTextFont*-- PROCEDURE RETURNS [MappedFontHandle];
MappedFont: --*SimpleTextFont*-- PROCEDURE [name: XString.Reader ^NIL]
RETURNS [MappedFontHandle];
MappedFontDescriptor: --*SimpleTextFont*-- TYPE;
MappedFontHandle: --*SimpleTextFont*-- TYPE = LONG POINTER TO
MappedFontDescriptor;
MapPList: --*Atom*-- PROCEDURE [atom: ATOM, proc: MapPListProc]
RETURNS [lastPair: RefPair];
MapPListProc: --*Atom*-- TYPE = PROCEDURE [RefPair] RETURNS [BOOLEAN];
Mark: --*ContainerCache*-- TYPE = LONG POINTER TO MarkObject;
MarkObject: --*ContainerCache*-- TYPE;
Match: --*Selection*-- PROCEDURE [pointer: ManagerData] RETURNS [match: BOOLEAN];
maxStringLength: --*Selection*-- CARDINAL = 200;
MaybeQuoted: --*XToken*-- PROCEDURE [
h: Handle, data: FilterState, filter: FilterProcType ^NonWhiteSpace,
isQuote: QuoteProcType ^Quote, skip: SkipMode ^whiteSpace,
temporary: BOOLEAN ^TRUE] RETURNS [value: XString.ReaderBody];
MeasureString: --*SimpleTextDisplay*-- PROCEDURE [
string: XString.Reader, lineWidth: CARDINAL ^177777B,
wordBreak: BOOLEAN ^TRUE, streakSuccession: StreakSuccession ^fromFirstChar,
font: SimpleTextFont.MappedFontHandle ^NIL]
RETURNS [width: CARDINAL, result: Result, rest: XString.ReaderBody];
MenuArray: --*MenuData*-- PROCEDURE [menu: MenuHandle]
RETURNS [array: ArrayHandle];
MenuEnumProc: --*StarWindowShell*-- TYPE = PROCEDURE [menu: MenuData.MenuHandle]
RETURNS [stop: BOOLEAN ^FALSE];
MenuHandle: --*MenuData*-- TYPE = LONG POINTER TO MenuObject;
MenuItemProc: --*PropertySheet*-- TYPE = PROCEDURE [
shell: StarWindowShell.Handle, formWindow: Window.Handle,
menuItem: MenuItemType, clientData: LONG POINTER] RETURNS [ok: BOOLEAN];
MenuItems: --*PropertySheet*-- TYPE = PACKED ARRAY MenuItemType OF
BooleanFalseDefault;
MenuItemType: --*PropertySheet*-- TYPE = {
done, apply, cancel, defaults, start, reset};
MenuObject: --*MenuData*-- TYPE = PrivateMenu;
MenuProc: --*MenuData*-- TYPE = PROCEDURE [
window: Window.Handle, menu: MenuHandle, itemData: LONG UNSPECIFIED];
MenuTitle: --*MenuData*-- PROCEDURE [menu: MenuHandle]
RETURNS [title: ItemHandle];
Messages: --*XMessage*-- TYPE = LONG DESCRIPTOR FOR ARRAY CARDINAL OF MsgEntry;
MessagesFromFile: --*XMessage*-- PROCEDURE [
fileName: LONG STRING, clientData: ClientData, proc: DestroyMsgsProc]
RETURNS [msgDomains: MsgDomains];
MessagesFromReference: --*XMessage*-- PROCEDURE [
file: NSFile.Reference, clientData: ClientData, proc: DestroyMsgsProc]
RETURNS [msgDomains: MsgDomains];
MinDimsChangeProc: --*FormWindow*-- TYPE = PROCEDURE [
window: Window.Handle, old: Window.Dims, new: Window.Dims];
MinusLandBitmapUnder: --*Window*-- TYPE [6];

MinusLandColor: --*Window*-- TYPE [1];
MinusLandCookieCutter: --*Window*-- TYPE [2];
Mode: --*TIPStar*-- TYPE = {normal, copy, move, sameAs};
ModeChangeProc: --*TIPStar*-- TYPE = PROCEDURE [
  old: Mode, new: Mode, clientData: LONG POINTER];
ModifySource: --*ContainerWindow*-- PROCEDURE [
  window: Window.Handle, proc: SourceModifyProc];
Months: --*XComSoftMessage*-- TYPE = Keys [january..december];
MoreFlavor: --*StarWindowShell*-- TYPE = {before, after};
MoreScrollProc: --*StarWindowShell*-- TYPE = PROCEDURE [
  sws: Handle, vertical: BOOLEAN, flavor: MoreFlavor, amount: CARDINAL];
MouseTransformerProc: --*Window*-- TYPE = PROCEDURE [Handle, Place]
  RETURNS [Handle, Place];
Move: --*Selection*-- PROCEDURE [v: ValueHandle, data: LONG POINTER];
MoveIntoWindow: --*Cursor*-- PROCEDURE [
  window: Window.Handle, place: Window.Place];
MoveMark: --*ContainerCache*-- PROCEDURE [mark: Mark, newIndex: CARDINAL];
MsgDomain: --*XMessage*-- TYPE = RECORD [
  applicationName: XString.ReaderBody, handle: Handle];
MsgDomains: --*XMessage*-- TYPE = LONG DESCRIPTOR FOR ARRAY CARDINAL OF MsgDomain;
MsgEntry: --*XMessage*-- TYPE = RECORD [
  msgKey: MsgKey,
  msg: XString.ReaderBody,
  translationNote: LONG STRING ^NIL,
  translatable: BOOLEAN ^TRUE,
  type: MsgType ^userMsg,
  id: MsgID];
MsgID: --*XMessage*-- TYPE = CARDINAL;
MsgKey: --*XMessage*-- TYPE = CARDINAL;
MsgKeyList: --*XMessage*-- TYPE = LONG DESCRIPTOR FOR ARRAY CARDINAL OF MsgKey;
MsgType: --*XMessage*-- TYPE = {
  userMsg, template, argList, menuItem, pSheetItem, commandItem, errorMsg,
  infoMsg, promptItem, windowMenuCommand, others};
MultiAttributeFormatProc: --*FileContainerSource*-- TYPE = PROCEDURE [
  containeeImpl: Containee.Implementation, containeeData: Containee.DataHandle,
  attrRecord: NSFile.Attributes, displayString: XString.Writer];
MultipleChoiceChangeProc: --*FormWindow*-- TYPE = PROCEDURE [
  window: Window.Handle, item: ItemKey, calledBecauseOf: ChangeReason,
  oldValue: LONG DESCRIPTOR FOR ARRAY CARDINAL OF ChoiceIndex,
  newValue: LONG DESCRIPTOR FOR ARRAY CARDINAL OF ChoiceIndex];
Multiply: --*XLReal*-- PROCEDURE [a: Number, b: Number] RETURNS [Number];
NameAndVersionColumn: --*FileContainerSourceExtra*-- PROCEDURE
  RETURNS [multipleAttributes FileContainerSource.ColumnContentsInfo];
NameColumn: --*FileContainerSource*-- PROCEDURE
  RETURNS [attribute ColumnContentsInfo];
NeededDims: --*FormWindow*-- PROCEDURE [window: Window.Handle]
  RETURNS [Window.Dims];
Negative: --*XLReal*-- PROCEDURE [Number] RETURNS [Number];
netAddr: --*BWSAttributeTypes*-- NSFile.ExtendedAttributeType = 10402B;
NetFormat: --*XFormat*-- TYPE = {octal, hex, productSoftware};
NetworkAddress: --*XFormat*-- PROCEDURE [
  h: Handle ^NIL, networkAddress: System.NetworkAddress, format: NetFormat];
networkName: --*BWSAttributeTypes*-- NSFile.ExtendedAttributeType = 10404B;
NetworkNumber: --*XFormat*-- PROCEDURE [
  h: Handle ^NIL, networkNumber: System.NetworkNumber, format: NetFormat];

New: --*Window*-- PROCEDURE [
  under: BOOLEAN ^FALSE, cookie: BOOLEAN ^FALSE, color: BOOLEAN ^FALSE,
  zone: UNCOUNTED ZONE ^LOOPHOLE[0]] RETURNS [Handle];
newIcon: --*StarDesktop*-- Atom.ATOM;
NewResolveBuffer: --*SimpleTextDisplay*-- PROCEDURE [words: CARDINAL]
  RETURNS [ResolveBuffer];
NewStandardCloseEverything: --*StarWindowShellExtra*-- PROCEDURE
  RETURNS [
    numberLeftOpen: CARDINAL ^0,
    lastNotClosed: StarWindowShell.Handle ^LOOPHOLE[0]];
NewWriterBody: --*XString*-- PROCEDURE [maxLength: CARDINAL, z: UNCOUNTED ZONE]
  RETURNS [WriterBody];
NextIntoProc: --*FormWindow*-- TYPE = PROCEDURE [
  window: Window.Handle, item: ItemKey];
NextOutOfProc: --*FormWindow*-- TYPE = PROCEDURE [
  window: Window.Handle, item: ItemKey];
nextPlace: --*StarDesktop*-- Window.Place;
nextTabStop: --*FormWindow*-- CARDINAL = 177777B;
NilData: --*XToken*-- SIGNAL;
nonQuote: --*XToken*-- XChar.Character = 0;
NonWhiteSpace: --*XToken*-- FilterProcType;
NopDestroyProc: --*Context*-- DestroyProcType;
NopFree: --*Selection*-- ValueFreeProc;
nopFreeValueProcs: --*Selection*-- READONLY LONG POINTER TO ValueProcs;
NormalTable: --*TIPStar*-- PROCEDURE RETURNS [TIP.Table];
noScrollData: --*StarWindowShell*-- ScrollData;
NoSuchAtom: --*Atom*-- ERROR;
NoSuchDependency: --*Event*-- ERROR;
not: --*XChar*-- Character = 177777B;
noTabStop: --*FormWindow*-- CARDINAL = 177776B;
NotAProfileFile: --*OptionFile*-- SIGNAL;
NotEq: --*XLReal*-- PROCEDURE [a: Number, b: Number] RETURNS [BOOLEAN];
Notes: --*XTime*-- TYPE = {
  normal, nozone, zonedGuessed, noTime, timeAndZoneGuessed};
Notify: --*Event*-- PROCEDURE [event: EventType, eventData: LONG POINTER ^NIL]
  RETURNS [veto: BOOLEAN];
NotifyProc: --*TIP*-- TYPE = PROCEDURE [window: Window.Handle, results: Results];
NSChar: --*XFormat*-- PROCEDURE [h: Handle ^NIL, char: NSString.Character];
NSLine: --*XFormat*-- PROCEDURE [
  h: Handle ^NIL, s: NSString.String, n: CARDINAL ^1];
NSString: --*XFormat*-- PROCEDURE [h: Handle ^NIL, s: NSString.String];
NSStringFromReader: --*XString*-- PROCEDURE [r: Reader, z: UNCOUNTED ZONE]
    RETURNS [ns: NSString.String];
NSStringObject: --*XFormat*-- PROCEDURE [s: LONG POINTER TO NSString.String]
  RETURNS [Object];
SStringProc: --*XFormat*-- FormatProc;
NthCharacter: --*XString*-- PROCEDURE [r: Reader, n: CARDINAL]
  RETURNS [c: Character];
null: --*Atom*-- ATOM;
null: --*XChar*-- Character = 0;
nullBox: --*Window*-- Box;
nullData: --*Containee*-- Data;
nullHandle: --*StarWindowShell*-- Handle;

nullItem: --*ContainerSource*-- ItemIndex = 177777B;
nullItemKey: --*FormWindow*-- ItemKey = 177777B;
nullKey: --*SoftKeys*-- CARDINAL = 177777B;
nullManager: --*TIP*-- Manager;
nullOption: --*ProductFactoring*-- Option;
nullPeriodicNotify: --*TIP*-- PeriodicNotify;
nullPicture: --*BlackKeys*-- bitmap Picture;
nullPlace: --*PropertySheet*-- Window.Place;
nullPrerequisite: --*ProductFactoring*-- Prerequisite;
nullReaderBody: --*XString*-- ReaderBody;
nullValue: --*Selection*-- Value;
nullWriterBody: --*XString*-- WriterBody;
Number: --*XFormat*-- PROCEDURE [
h: Handle ˆNIL, n: LONG UNSPECIFIED, format: NumberFormat];
Number: --*XLReal*-- TYPE [4];
Number: --*XToken*-- PROCEDURE [
h: Handle, radix: CARDINAL, signalOnError: BOOLEAN ˆTRUE]
RETURNS [u: LONG UNSPECIFIED];
NumberFormat: --*XFormat*-- TYPE = RECORD [
base: [2..36] ˆ12,
zerofill: BOOLEAN ˆFALSE,
signed: BOOLEAN ˆFALSE,
columns: [0..255] ˆ0];
NumberOfItems: --*FormWindow*-- PROCEDURE [window: Window.Handle]
RETURNS [CARDINAL];
numberOfKeys: --*SoftKeys*-- CARDINAL = 8;
NumberToPair: --*XLReal*-- PROCEDURE [n: Number, digits: [1..13]]
RETURNS [negative: BOOLEAN, exp: INTEGER, mantissa: Digits];
Numeric: --*XToken*-- FilterProcType;
Object: --*ContainerCache*-- TYPE;
Object: --*Cursor*-- TYPE = RECORD [info: Info, array: UserTerminal.CursorArray];
Object: --*Window*-- TYPE [19];
Object: --*XFormat*-- TYPE = RECORD [
  proc: FormatProc,
  context: XString.Context ˆLOOPHOLE[0],
  data: ClientData ˆNIL];
Object: --*XMessage*-- TYPE;
Object: --*XToken*-- TYPE = MACHINE DEPENDENT RECORD [
getChar(0:0..31): GetCharProcType, break(2:0..15): XChar.Character ˆ0];
ObscuredBySibling: --*Window*-- PROCEDURE [Handle] RETURNS [BOOLEAN];
Octal: --*XFormat*-- PROCEDURE [h: Handle ˆNIL, n: LONG UNSPECIFIED];
Octal: --*XToken*-- PROCEDURE [h: Handle, signalOnError: BOOLEAN ˆTRUE]
  RETURNS [c: LONG CARDINAL];
OctalFormat: --*XFormat*-- NumberFormat;
oldDateSent: --*BWSAttributeTypes*-- NSFile.ExtendedAttributeType = 10413B;
Open: --*Catalog*-- PROCEDURE [
  catalogType: NSFile.Type, session: NSFile.Session ˆLOOPHOLE[0]]
  RETURNS [catalog: NSFile.Handle];
Opportunity: --*Undo*-- Proc;
Option: --*ProductFactoring*-- TYPE = RECORD [
  product: Product, productOption: ProductOption];
Options: --*FileContainerSource*-- TYPE = RECORD [readOnly: BOOLEAN ˆFALSE];
optionSheetDefaultMenu: --*PropertySheet*-- MenuItems;
outbasketPSData: --*BWSAttributeTypes*-- NSFile.ExtendedAttributeType = 10410B;
OutlineOrHighlight: --*FormWindow*-- TYPE = {outline, highlight};

OutlineThisKey: --*SoftKeys*-- PROCEDURE [
  window: Window.Handle, key: CARDINAL ˆnullKey];
Overflow: --*XString*-- SIGNAL;
owner: --*BWSAttributeTypes*-- NSFile.ExtendedAttributeType = 10377B;
Pack: --*XTime*-- PROCEDURE [unpacked: Unpacked, useSystemLTP: BOOLEAN ˆTRUE]
  RETURNS [time: System.GreenwichMeanTime];
Packed: --*XTime*-- TYPE = System.GreenwichMeanTime;
paintFlags: --*Display*-- BitBltFlags;
paintGrayFlags: --*Display*-- BitBltFlags;
Pair: --*Atom*-- TYPE = RECORD [prop: ATOM, value: RefAny];
PairToNumber: --*XLReal*-- PROCEDURE [
  negative: BOOLEAN, exp: INTEGER, mantissa: Digits] RETURNS [n: Number];
Parallelogram: --*Display*-- TYPE = RECORD [
  x: Interpolator, y: INTEGER, w: NATURAL, h: NATURAL];
ParseChoiceItemMessage: --*FormWindowMessageParse*-- PROCEDURE [
  choiceItemMessage: XString.Reader, zone: UNCOUNTED ZONE]
  RETURNS [choiceItems: FormWindow.ChoiceItems];
ParseReader: --*XTime*-- PROCEDURE [
  r: XString.Reader, treatNumbersAs: TreatNumbersAs ˆdayMonthYear]
  RETURNS [time: System.GreenwichMeanTime, notes: Notes, length: CARDINAL];
ParseWithTemplate: --*XTime*-- PROCEDURE [
  r: XString.Reader, template: XString.Reader]
  RETURNS [time: System.GreenwichMeanTime, notes: Notes, length: CARDINAL];
PeekForFlushness: --*SimpleTextDisplay*-- PROCEDURE [
  requestedFlushness: Flushness, string: XString.Reader] RETURNS [Flushness];
PeekForStreakSuccession: --*SimpleTextDisplay*-- PROCEDURE [
  requestedStreakSuccession: StreakSuccession, string: XString.Reader]
  RETURNS [StreakSuccession];
PeriodicNotify: --*TIP*-- TYPE [1];
Permanent: --*BWSZone*-- PROCEDURE RETURNS [UNCOUNTED ZONE];
permanent: --*BWSZone*-- UNCOUNTED ZONE;
PFonts: --*ProductFactoringProducts*-- Product = 4;
Pi: --*XLReal*-- PROCEDURE RETURNS [Number];
Picture: --*BlackKeys*-- TYPE = RECORD [
  variant: SELECT type: PictureType FROM
    bitmap => [bitmap: LONG POINTER], text => [text: XString.Reader], ENDCASE];
PictureAction: --*BlackKeys*-- TYPE = {acquire, release};
PictureProc: --*BlackKeys*-- TYPE = PROCEDURE [
  keyboard: Keyboard, action: PictureAction]
  RETURNS [picture: Picture ˆnullPicture, geometry: GeometryTable ˆNIL];
PictureProc: --*Containee*-- TYPE = PROCEDURE [
  data: DataHandle, window: Window.Handle, box: Window.Box, old: PictureState,
  new: PictureState];
PictureReal: --*XLReal*-- PROCEDURE [
  h: XFormat.Handle ˆNIL, r: Number, template: XString.Reader];
PictureState: --*Containee*-- TYPE = {
  garbage, normal, highlighted, ghost, reference, referenceHighlighted};
PictureType: --*BlackKeys*-- TYPE = {bitmap, text};
Piece: --*XString*-- PROCEDURE [r: Reader, firstChar: CARDINAL, nChars: CARDINAL]
  RETURNS [piece: ReaderBody, endContext: Context];
Place: --*Window*-- TYPE = UserTerminal.Coordinate;
Placeholder: --*TIPStar*-- TYPE = {
  mouseActions, keyOverrides, softKeys, keyboardSpecific, blackKeys, sideKeys,
  backstopSpecialFocus};
Point: --*Display*-- PROCEDURE [window: Handle, point: Window.Place];
Pop: --*StarWindowShell*-- PROCEDURE [popee: Handle] RETURNS [Handle];

```
PopOrSwap: --StarWindowShell-- TYPE = {pop, swap};
PoppedProc: --StarWindowShell-- TYPE = PROCEDURE [
   popped: Handle, newShell: Handle, popOrSwap: PopOrSwap ^pop];
PopTable: --TIPStar-- PROCEDURE [Placeholder, TIP.Table];
Popup: --PopupMenu-- PROCEDURE [
   menu: MenuData.MenuHandle, clients: Window.Handle, showTitle: BOOLEAN ^TRUE,
   place: Window.Place ^LOOPHOLE[37777777777B]];
Post: --Attention-- PROCEDURE [
   s: XString.Reader, clear: BOOLEAN ^TRUE, beep: BOOLEAN ^FALSE,
   blink: BOOLEAN ^FALSE];
Post: --MessageWindow-- PROCEDURE [
   window: Window.Handle, r: XString.Reader, clear: BOOLEAN ^TRUE];
PostAndConfirm: --Attention-- PROCEDURE [
   s: XString.Reader, clear: BOOLEAN ^TRUE,
   confirmChoices: ConfirmChoices ^xxx, timeout: Process.Ticks ^dontTimeout,
   beep: BOOLEAN ^FALSE, blink: BOOLEAN ^FALSE]
   RETURNS [confirmed: BOOLEAN, timedOut: BOOLEAN];
PostSticky: --Attention-- PROCEDURE [
   s: XString.Reader, clear: BOOLEAN ^TRUE, beep: BOOLEAN ^FALSE,
   blink: BOOLEAN ^FALSE];
PostSTRING: --MessageWindow-- PROCEDURE [
   window: Window.Handle, s: LONG STRING, clear: BOOLEAN ^TRUE];
Power: --XLReal-- PROCEDURE [base: Number, exponent: Number] RETURNS [Number];
Prerequisite: --ProductFactoring-- TYPE = RECORD [
   prerequisiteSpec: BOOLEAN ^FALSE, option: Option];
printingLigatures: --XCharSet360-- XCharSets.Sets = LOOPHOLE[240];
PrivateItem: --MenuData-- TYPE = PRIVATE RECORD [
   proc: MenuProc,
   nameWidth: NATURAL,
   nameBytes: NATURAL,
   body: SELECT hasItemData: BOOLEAN FROM
      FALSE => [name: PACKED SEQUENCE COMPUTED CARDINAL OF Environment.Byte],
      TRUE => [
         itemData: LONG UNSPECIFIED,
         name: PACKED SEQUENCE COMPUTED CARDINAL OF Environment.Byte],
      ENDCASE];
PrivateMenu: --MenuData-- TYPE = PRIVATE RECORD [
   zone: UNCOUNTED ZONE,
   swapItemProc: SwapItemProc,
   title: ItemHandle ^NIL,
   array: ArrayHandle ^xxx,
   arrayAllocatedItemHandles: NATURAL ^0,
   itemsInMenusZone: BOOLEAN ^FALSE];
Problem: --SimpleTextFont-- SIGNAL [code: ProblemCode];
ProblemCode: --SimpleTextFont-- TYPE = {
   badFont, clientCharacterCodesExhausted, clientCharacterBitsExhausted};
Proc: --Undo-- TYPE = PROCEDURE [
   undoProc: PROCEDURE [LONG POINTER], destroyProc: PROCEDURE [LONG POINTER],
   data: LONG POINTER, size: CARDINAL ^0];
Procedures: --ContainerSource-- TYPE = LONG POINTER TO ProceduresObject;
ProceduresObject: --ContainerSource-- TYPE = RECORD [
   actOn: ActOnProc,
   canYouTake: CanYouTakeProc,
   columnCount: ColumnCountProc,
   convertItem: ConvertItemProc,
   deleteItems: DeleteItemsProc,
   getLength: GetLengthProc,
   itemGeneric: ItemGenericProc,
   stringOfItem: StringOfItemProc,
   take: TakeProc];
Product: --ProductFactoring-- TYPE = CARDINAL [0..15];
Product: --ProductFactoringProducts-- TYPE = ProductFactoring.Product;
Product: --ProductFactoringProductsExtras-- TYPE = ProductFactoring.Product;
ProductOption: --ProductFactoring-- TYPE = CARDINAL [0..27];
propertySheetDefaultMenu: --PropertySheet-- MenuItems;
prototypeCatalog: --BWSFileTypes-- NSFile.Type = 1;
PublicZone: --MenuData-- PROCEDURE RETURNS [UNCOUNTED ZONE];
PurgeOldVersions: --Prototype-- PROCEDURE [
   type: NSFile.Type, current: Version, subtype: Subtype ^0];
Push: --BlackKeys-- PROCEDURE [keyboard: Keyboard];
Push: --SoftKeys-- PROCEDURE [
   table: TIP.Table ^NIL, notifyProc: TIP.NotifyProc ^NIL,
   labels: Labels ^xxx, highlightedKey: CARDINAL ^nullKey,
   outlinedKey: CARDINAL ^nullKey] RETURNS [window: Window.Handle];
Push: --StarWindowShell-- PROCEDURE [
   newShell: Handle, topOfStack: Handle ^LOOPHOLE[0],
   poppedProc: PoppedProc ^NIL];
PushedMe: --StarWindowShellExtra-- PROCEDURE [pushee: StarWindowShell.Handle]
   RETURNS [pusher: StarWindowShell.Handle];
PushedOnMe: --StarWindowShellExtra-- PROCEDURE [pusher: StarWindowShell.Handle]
   RETURNS [pushee: StarWindowShell.Handle];
PushTable: --TIPStar-- PROCEDURE [Placeholder, TIP.Table];
PutProp: --Atom-- PROCEDURE [onto: ATOM, pair: Pair];
Query: --Selection-- PROCEDURE [
   targets: LONG DESCRIPTOR FOR ARRAY CARDINAL OF QueryElement];
QueryElement: --Selection-- TYPE = RECORD [
   target: Target, enumeration: BOOLEAN ^FALSE, difficulty: Difficulty ^NULL];
Quote: --XToken-- QuoteProcType;
QuoteProcType: --XToken-- TYPE = PROCEDURE [c: XChar.Character]
   RETURNS [closing: XChar.Character];
Reader: --XFormat-- PROCEDURE [h: Handle ^NIL, r: XString.Reader];
Reader: --XString-- TYPE = LONG POINTER TO ReaderBody;
ReaderBody: --XFormat-- PROCEDURE [h: Handle ^NIL, rb: XString.ReaderBody];
ReaderBody: --XString-- TYPE = PRIVATE MACHINE DEPENDENT RECORD [
   context(0:0..15): Context,
   limit(1:0..15): CARDINAL,
   offset(2:0..15): CARDINAL,
   bytes(3:0..31): ReadOnlyBytes];
ReaderFromWriter: --XString-- PROCEDURE [w: Writer] RETURNS [Reader];
ReaderInfo: --XString-- PROCEDURE [r: Reader]
   RETURNS [context: Context, startsWith377B: BOOLEAN];
ReaderToHandle: --XToken-- PROCEDURE [r: XString.Reader] RETURNS [h: Handle];
ReaderToNumber: --XLReal-- PROCEDURE [r: XString.Reader] RETURNS [Number];
ReaderToNumber: --XString-- PROCEDURE [
   r: Reader, radix: CARDINAL ^10, signed: BOOLEAN ^FALSE]
   RETURNS [LONG INTEGER];
ReadNumber: --XLReal-- PROCEDURE [
   get: PROCEDURE RETURNS [XChar.Character],
   putback: PROCEDURE [XChar.Character]] RETURNS [Number];
ReadOnlyBytes: --XString-- TYPE = LONG POINTER TO READONLY ByteSequence;
```

RebuildItem: *--FileContainerSourceExtra2--* PROCEDURE [
  source: ContainerSource.Handle, item: ContainerSource.ItemIndex];
Reconversion: *--Selection--* SIGNAL [target: Target, zone: UNCOUNTED ZONE]
  RETURNS [Value];
ReconvertDuringEnumerate: *--Selection--* PROCEDURE [
  target: Target, zone: UNCOUNTED ZONE ^LOOPHOLE[0]] RETURNS [Value];
RefAny: *--Atom--* TYPE = LONG POINTER;
referencedType: *--BWSAttributeTypes--* NSFile.ExtendedAttributeType = 10401B;
RefPair: *--Atom--* TYPE = LONG POINTER TO READONLY Pair;
refParentID: *--BWSAttributeTypes--* NSFile.ExtendedAttributeType = 10403B;
refparentTime: *--BWSAttributeTypes--* NSFile.ExtendedAttributeType = 10405B;
RegisterClientKeyboards: *--KeyboardKey--* PROCEDURE [
  wantSystemKeyboards: BOOLEAN ^TRUE,
  SPECIALKeyboard: BlackKeys.Keyboard ^NIL,
  keyboards: LONG DESCRIPTOR FOR ARRAY CARDINAL OF BlackKeys.KeyboardObject ^
    xxx];
RegisterMessages: *--XMessage--* PROCEDURE [
  h: Handle, messages: Messages, stringBodiesAreReal: BOOLEAN];
Relation: *--XString--* TYPE = {less, equal, greater};
Release: *--Context--* PROCEDURE [type: Type, window: Window.Handle];
Remainder: *--XLReal--* PROCEDURE [a: Number, b: Number] RETURNS [Number];
remoteName: *--BWSAttributeTypes--* NSFile.ExtendedAttributeType = 10406B;
Remove: *--BlackKeys--* PROCEDURE [keyboard: Keyboard];
Remove: *--SoftKeys--* PROCEDURE [window: Window.Handle];
RemoveClientKeyboards: *--KeyboardKey--* PROCEDURE;
RemoveDependency: *--Event--* PROCEDURE [dependency: Dependency];
RemoveFromSystemKeyboards: *--KeyboardKey--* PROCEDURE [
  keyboard: BlackKeys.Keyboard];
RemoveFromTree: *--Window--* PROCEDURE [Handle];
RemoveItemFromLine: *--FormWindow--* PROCEDURE [
  window: Window.Handle, item: ItemKey, line: Line, repaint: BOOLEAN ^TRUE];
RemoveMenuItem: *--Attention--* PROCEDURE [item: MenuData.ItemHandle];
RemoveProp: *--Atom--* PROCEDURE [onto: ATOM, prop: ATOM];
Repaint: *--FormWindow--* PROCEDURE [window: Window.Handle];
RepaintField: *--SimpleTextEdit--* PROCEDURE [f: Field];
Replace: *--StarWindowShellExtra--* PROCEDURE [
  new: StarWindowShell.Handle, old: StarWindowShell.Handle];
ReplaceChars: *--SimpleTextEdit--* PROCEDURE [
  f: Field, firstChar: CARDINAL, nChars: CARDINAL, r: XString.Reader,
  endContext: XString.Context ^LOOPHOLE[255], repaint: BOOLEAN ^TRUE];
replaceFlags: *--Display--* BitBltFlags;
replaceGrayFlags: *--Display--* BitBltFlags;
ReplaceItem: *--ContainerCache--* PROCEDURE [
  cache: Handle, item: CARDINAL, addData: AddData] RETURNS [handle: ItemHandle];
ReplacePiece: *--XString--* PROCEDURE [
  w: Writer, firstChar: CARDINAL, nChars: CARDINAL, r: Reader,
  endContext: Context ^unknownContext];
RequestorData: *--Selection--* TYPE = LONG POINTER;
ResetAllChanged: *--FormWindow--* PROCEDURE [window: Window.Handle];
ResetCache: *--ContainerCache--* PROCEDURE [Handle];
ResetChanged: *--FormWindow--* PROCEDURE [window: Window.Handle, item: ItemKey];
ResetUserAbort: *--TIP--* PROCEDURE [Window.Handle];
ResolveBuffer: *--SimpleTextDisplay--* TYPE = LONG DESCRIPTOR FOR ARRAY [0..0) OF
  CARDINAL;
Restore: *--FormWindow--* PROCEDURE [window: Window.Handle];

```
Restore: --Selection-- PROCEDURE [
   saved: Saved, mark: BOOLEAN ^TRUE, unmark: BOOLEAN ^TRUE];
Result: --SimpleTextDisplay-- TYPE = {normal, margin, stop};
ResultObject: --TIP-- TYPE = RECORD [
   next: Results,
   body: SELECT type: * FROM
      atom => [a: ATOM],
      bufferedChar => NULL,
      coords => [place: Window.Place],
      int => [i: LONG INTEGER],
      key => [key: KeyName, downUp: DownUp],
      nop => NULL,
      string => [rb: XString.ReaderBody],
      time => [time: System.Pulses],
      ENDCASE];
Results: --TIP-- TYPE = LONG POINTER TO ResultObject;
ResultsWanted: --TIPX-- TYPE = PROCEDURE [
   window: Window.Handle, table: TIP.Table ^NIL, results: TIP.Results]
   RETURNS [wanted: BOOLEAN];
ReturnTicket: --Containee-- PROCEDURE [ticket: Ticket];
ReturnToNotifier: --TIP-- ERROR [string: XString.Reader];
ReverseLop: --XString-- PROCEDURE [
   r: Reader, endContext: LONG POINTER TO Context,
   backScan: BackScanClosure ^xxx] RETURNS [c: Character];
ReverseMap: --XString-- PROCEDURE [r: Reader, proc: MapCharProc]
   RETURNS [c: Character];
Roadblock: --Undo-- PROCEDURE [XString.Reader];
root: --BWSFileTypes-- NSFile.Type = 10477B;
Root: --Window-- PROCEDURE RETURNS [Handle];
Root: --XLReal-- PROCEDURE [index: Number, arg: Number] RETURNS [Number];
rootWindow: --Window-- READONLY Handle;
Run: --XString-- PROCEDURE [r: Reader] RETURNS [run: ReaderBody];
Save: --FormWindow-- PROCEDURE [window: Window.Handle];
SaveAndSet: --Selection-- PROCEDURE [
   pointer: ManagerData, conversion: ConvertProc, actOn: ActOnProc,
   unmark: BOOLEAN ^TRUE] RETURNS [old: Saved];
Saved: --Selection-- TYPE [6];
Scan: --XString-- PROCEDURE [
   r: Reader, break: BreakTable, option: BreakCharOption]
   RETURNS [breakChar: Character, front: ReaderBody];
ScanForCharacter: --XString-- PROCEDURE [
   r: Reader, char: Character, option: BreakCharOption]
   RETURNS [breakChar: Character, front: ReaderBody];
ScrollData: --StarWindowShell-- TYPE = RECORD [
   displayHorizontal: BOOLEAN ^FALSE,
   displayVertical: BOOLEAN ^FALSE,
   arrowScroll: ArrowScrollProc ^NIL,
   thumbScroll: ThumbScrollProc ^NIL,
   moreScroll: MoreScrollProc ^NIL];
SectionEnumProc: --OptionFile-- TYPE = PROCEDURE [section: XString.Reader]
   RETURNS [stop: BOOLEAN ^FALSE];
SelectItem: --ContainerWindow-- PROCEDURE [
   window: Window.Handle, item: ContainerSource.ItemIndex];
SelectReference: --StarDesktop-- PROCEDURE [reference: NSFile.Reference]
   RETURNS [ok: BOOLEAN];
SemiPermanent: --BWSZone-- PROCEDURE RETURNS [UNCOUNTED ZONE];
```

semiPermanent: --*BWSZone*-- UNCOUNTED ZONE;
Services2: --*ProductFactoringProductsExtras*-- Product = 8;
Services: --*ProductFactoringProducts*-- Product = 1;
Set: --*Context*-- PROCEDURE [type: Type, data: Data, window: Window.Handle];
Set: --*Cursor*-- PROCEDURE [type: Defined];
Set: --*Selection*-- PROCEDURE [
  pointer: ManagerData, conversion: ConvertProc, actOn: ActOnProc];
Set: --*XChar*-- PROCEDURE [c: Character] RETURNS [set: Environment.Byte];
SetAdjustProc: --*StarWindowShell*-- PROCEDURE [sws: Handle, proc: AdjustProc]
  RETURNS [old: AdjustProc];
SetAllChanged: --*FormWindow*-- PROCEDURE [window: Window.Handle];
SetAttention: --*TIP*-- PROCEDURE [
  window: Window.Handle, attention: AttentionProc];
SetBackStopInputFocus: --*TIP*-- PROCEDURE [window: Window.Handle];
SetBitmapUnder: --*Window*-- PROCEDURE [
  window: Handle, pointer: LONG POINTER ˆNIL,
  underChanged: UnderChangedProc ˆNIL,
  mouseTransformer: MouseTransformerProc ˆNIL] RETURNS [LONG POINTER];
SetBodyWindowJustFits: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, yes: BOOLEAN];
SetBOOLEAN: --*AtomicProfile*-- PROCEDURE [atom: Atom.ATOM, boolean: BOOLEAN];
SetBooleanItemValue: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, newValue: BOOLEAN,
  repaint: BOOLEAN ˆTRUE];
SetBottomPusheeCommands: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, commands: MenuData.MenuHandle];
SetCachedName: --*Containee*-- PROCEDURE [
  data: DataHandle, newName: XString.Reader];
SetCachedType: --*Containee*-- PROCEDURE [data: DataHandle, newType: NSFile.Type];
SetChanged: --*FormWindow*-- PROCEDURE [window: Window.Handle, item: ItemKey];
SetCharTranslator: --*TIP*-- PROCEDURE [table: Table, new: CharTranslator]
  RETURNS [old: CharTranslator];
SetChild: --*Window*-- PROCEDURE [window: Handle, newChild: Handle]
  RETURNS [oldChild: Handle];
SetChoiceItemValue: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, newValue: ChoiceIndex,
  repaint: BOOLEAN ˆTRUE];
SetClearingRequired: --*Window*-- PROCEDURE [window: Handle, required: BOOLEAN]
  RETURNS [old: BOOLEAN];
SetContainee: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, file: Containee.DataHandle];
SetDecimalItemValue: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, newValue: XLReal.Number,
  repaint: BOOLEAN ˆTRUE];
SetDefaultImplementation: --*Containee*-- PROCEDURE [Implementation]
  RETURNS [Implementation];
SetDefaultOutputSink: --*XFormat*-- PROCEDURE [new: Object] RETURNS [old: Object];
SetDesktopProc: --*IdleControl*-- PROCEDURE [
  atom: Atom.ATOM, desktop: DesktopProc];
SetDims: --*SimpleTextEdit*-- PROCEDURE [f: Field, dims: Window.Dims];
SetDisplayBackgroundProc: --*StarDesktop*-- PROCEDURE [PROCEDURE [Window.Handle]];
SetDisplayProc: --*Window*-- PROCEDURE [Handle, DisplayProc]
  RETURNS [DisplayProc];
SetFixedHeight: --*SimpleTextEdit*-- PROCEDURE [f: Field, fixedHeight: BOOLEAN];

SetFlushness: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, new: Flushness]
  RETURNS [old: Flushness];
SetFlushness: --*SimpleTextEdit*-- PROCEDURE [
  f: Field, new: SimpleTextDisplay.Flushness]
  RETURNS [old: SimpleTextDisplay.Flushness];
SetFont: --*SimpleTextEdit*-- PROCEDURE [
  f: Field, font: SimpleTextFont.MappedFontHandle ^NIL];
SetGlobalChangeProc: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, proc: GlobalChangeProc]
  RETURNS [old: GlobalChangeProc];
SetGreeterProc: --*IdleControl*-- PROCEDURE [new: GreeterProc]
  RETURNS [old: GreeterProc];
SetHost: --*StarWindowShell*-- PROCEDURE [sws: Handle, host: Handle]
  RETURNS [old: Handle];
SetImplementation: --*Containee*-- PROCEDURE [NSFile.Type, Implementation]
  RETURNS [Implementation];
SetImplementation: --*Undo*-- PROCEDURE [Implementation] RETURNS [Implementation];
SetInputFocus: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, beforeChar: CARDINAL ^177777B];
SetInputFocus: --*SimpleTextEdit*-- PROCEDURE [
  f: Field, beforeChar: CARDINAL ^177777B];
SetInputFocus: --*TIP*-- PROCEDURE [
  w: Window.Handle, takesInput: BOOLEAN, newInputFocus: LosingFocusProc ^NIL,
  clientData: LONG POINTER ^NIL];
SetIntegerItemValue: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, newValue: LONG INTEGER,
  repaint: BOOLEAN ^TRUE];
SetIsCloseLegalProc: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, proc: IsCloseLegalProc];
SetItemBox: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, box: Window.Box];
SetItemNameWidth: --*MenuData*-- PROCEDURE [item: ItemHandle, width: CARDINAL];
SetItemWidth: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey, width: CARDINAL];
SetKeyboard: --*KeyboardKey*-- PROCEDURE [keyboard: BlackKeys.Keyboard];
SetLimitProc: --*StarWindowShell*-- PROCEDURE [sws: Handle, proc: LimitProc]
  RETURNS [old: LimitProc];
SetLONGINTEGER: --*AtomicProfile*-- PROCEDURE [
  atom: Atom.ATOM, int: LONG INTEGER];
SetManager: --*TIP*-- PROCEDURE [new: Manager] RETURNS [old: Manager];
SetMark: --*ContainerCache*-- PROCEDURE [cache: Handle, index: CARDINAL]
  RETURNS [mark: Mark];
SetMiddlePusheeCommands: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, commands: MenuData.MenuHandle];
SetMode: --*TIPStar*-- PROCEDURE [
  mode: Mode, modeChangeProc: ModeChangeProc ^NIL,
  clientData: LONG POINTER ^NIL] RETURNS [old: Mode];
SetMultipleChoiceItemValue: --*FormWindow*-- PROCEDURE [
  window: Window.Handle, item: ItemKey,
  newValues: LONG DESCRIPTOR FOR ARRAY CARDINAL OF ChoiceIndex,
  repaint: BOOLEAN ^TRUE];
SetName: --*StarWindowShell*-- PROCEDURE [sws: Handle, name: XString.Reader];
SetNamePicture: --*StarWindowShell*-- PROCEDURE [
  sws: Handle, picture: XString.Character];

SetNextOutOfProc: *--FormWindow--* PROCEDURE [
window: Window.Handle, item: ItemKey, nextOutOfProc: NextOutOfProc]
RETURNS [old: NextOutOfProc];
SetNotifyProc: *--TIP--* PROCEDURE [window: Window.Handle, notify: NotifyProc]
RETURNS [oldNotify: NotifyProc];
SetNotifyProcForTable: *--TIP--* PROCEDURE [table: Table, notify: NotifyProc]
RETURNS [oldNotify: NotifyProc];
SetParent: *--Window--* PROCEDURE [window: Handle, newParent: Handle]
RETURNS [oldParent: Handle];
SetPlace: *--SimpleTextEdit--* PROCEDURE [f: Field, place: Window.Place];
SetPreferredDims: *--StarWindowShell--* PROCEDURE [
sws: Handle, dims: Window.Dims];
SetPreferredInteriorDims: *--StarWindowShellExtra2--* PROCEDURE [
sws: StarWindowShell.Handle, dims: Window.Dims];
SetPreferredPlace: *--StarWindowShell--* PROCEDURE [
sws: Handle, place: Window.Place];
SetReadOnly: *--FormWindow--* PROCEDURE [
window: Window.Handle, item: ItemKey, readOnly: BOOLEAN]
RETURNS [old: BOOLEAN];
SetReadOnly: *--SimpleTextEdit--* PROCEDURE [f: Field, readOnly: BOOLEAN]
RETURNS [old: BOOLEAN];
SetReadOnly: *--StarWindowShell--* PROCEDURE [sws: Handle, yes: BOOLEAN];
SetRegularCommands: *--StarWindowShell--* PROCEDURE [
sws: Handle, commands: MenuData.MenuHandle];
Sets: *--XCharSets--* TYPE = MACHINE DEPENDENT{
latin, firstUnused1, lastUnused1(32), jisSymbol1, jisSymbol2, extendedLatin,
hiragana, katakana, greek, cyrillic, firstUserKanji1, lastUserKanji1(47),
firstLevel1Kanji, lastLevel1Kanji(79), firstLevel2Kanji, lastLevel2Kanji(115),
jSymbol3, firstUserKanji2, lastUserKanji2(126), firstUnused2,
lastUnused2(160), firstReserved1, lastReserved1(223), arabic, hebrew,
firstReserved2, lastReserved2(237), generalSymbols2, generalSymbols1,
firstRendering, lastRendering(253), userDefined, selectCode};
SetScrollData: *--StarWindowShell--* PROCEDURE [sws: Handle, new: ScrollData]
RETURNS [old: ScrollData];
SetSelection: *--FormWindow--* PROCEDURE [
window: Window.Handle, item: ItemKey, firstChar: CARDINAL ^0,
lastChar: CARDINAL ^177777B];
SetSelection: *--SimpleTextEdit--* PROCEDURE [
f: Field, firstChar: CARDINAL ^0, lastChar: CARDINAL ^177777B];
SetShowKeyboardProc: *--KeyboardKey--* PROCEDURE [ShowKeyboardProc];
SetSibling: *--Window--* PROCEDURE [window: Handle, newSibling: Handle]
RETURNS [oldSibling: Handle];
SetSleeps: *--StarWindowShellExtra--* PROCEDURE [
sws: StarWindowShell.Handle, sleeps: BOOLEAN] RETURNS [old: BOOLEAN];
SetSource: *--ContainerWindow--* PROCEDURE [
window: Window.Handle, newSource: ContainerSource.Handle]
RETURNS [oldSource: ContainerSource.Handle];
SetState: *--StarWindowShell--* PROCEDURE [sws: Handle, state: State];
SetStreakSuccession: *--FormWindow--* PROCEDURE [
window: Window.Handle, item: ItemKey, new: StreakSuccession]
RETURNS [old: StreakSuccession];
SetStreakSuccession: *--SimpleTextEdit--* PROCEDURE [
f: Field, new: SimpleTextDisplay.StreakSuccession]
RETURNS [old: SimpleTextDisplay.StreakSuccession];
SetString: *--AtomicProfile--* PROCEDURE [
atom: Atom.ATOM, string: XString.Reader, immutable: BOOLEAN ^FALSE];

SetSwapItemProc: --*MenuData*-- PROCEDURE [menu: MenuHandle, new: SwapItemProc]
RETURNS [old: SwapItemProc];
SetTable: --*TIP*-- PROCEDURE [window: Window.Handle, table: Table]
RETURNS [oldTable: Table];
SetTableAndNotifyProc: --*TIP*-- PROCEDURE [
window: Window.Handle, table: Table ^NIL, notify: NotifyProc ^NIL];
SetTableLink: --*TIP*-- PROCEDURE [from: Table, to: Table] RETURNS [old: Table];
SetTableOpacity: --*TIP*-- PROCEDURE [table: Table, opaque: BOOLEAN]
RETURNS [oldOpaque: BOOLEAN];
SetTabStops: --*FormWindow*-- PROCEDURE [
window: Window.Handle, tabStops: TabStops];
SetTextItemValue: --*FormWindow*-- PROCEDURE [
window: Window.Handle, item: ItemKey, newValue: XString.Reader,
repaint: BOOLEAN ^TRUE];
SetTopPusheeCommands: --*StarWindowShell*-- PROCEDURE [
sws: Handle, commands: MenuData.MenuHandle];
SetTransitionProc: --*StarWindowShell*-- PROCEDURE [
sws: Handle, new: TransitionProc] RETURNS [old: TransitionProc];
SetUseBadPhosphor: --*Window*-- PROCEDURE [Handle, BOOLEAN] RETURNS [BOOLEAN];
SetUserAbort: --*TIP*-- PROCEDURE [Window.Handle];
SetValue: --*SimpleTextEdit*-- PROCEDURE [
f: Field, string: XString.Reader, repaint: BOOLEAN ^TRUE];
SetVisibility: --*FormWindow*-- PROCEDURE [
window: Window.Handle, item: ItemKey, visibility: Visibility,
repaint: BOOLEAN ^TRUE];
SetWindowItemSize: --*FormWindow*-- PROCEDURE [
window: Window.Handle, windowItemKey: ItemKey, newSize: Window.Dims];
ShellEnumProc: --*StarWindowShell*-- TYPE = PROCEDURE [sws: Handle]
RETURNS [stop: BOOLEAN ^FALSE];
ShellFromChild: --*StarWindowShell*-- PROCEDURE [child: Window.Handle]
RETURNS [Handle];
ShellType: --*StarWindowShell*-- TYPE = MACHINE DEPENDENT{
regular, keyboard, psheet, attention, static, last(15)};
Shift: --*Display*-- PROCEDURE [
window: Handle, box: Window.Box, newPlace: Window.Place];
ShiftState: --*KeyboardWindow*-- TYPE = {None, One, Two, Both};
ShortLifetime: --*BWSZone*-- PROCEDURE RETURNS [UNCOUNTED ZONE];
shortLifetime: --*BWSZone*-- UNCOUNTED ZONE;
ShowKeyboardProc: --*KeyboardKey*-- TYPE = PROCEDURE;
Signal: --*Containee*-- SIGNAL [
msg: XString.Reader ^NIL, error: ERROR ^NIL, errorData: LONG POINTER ^NIL];
Signal: --*ContainerSource*-- SIGNAL [
code: ErrorCode, msg: XString.Reader ^NIL, error: ERROR ^NIL,
errorData: LONG POINTER ^NIL];
SimpleDestroyProc: --*Context*-- DestroyProcType;
Sin: --*XLReal*-- PROCEDURE [radians: Number] RETURNS [sin: Number];
SizeColumn: --*FileContainerSource*-- PROCEDURE
RETURNS [multipleAttributes ColumnContentsInfo];
Skip: --*XToken*-- PROCEDURE [
h: Handle, data: FilterState, filter: FilterProcType,
skipInClass: BOOLEAN ^TRUE];
SkipMode: --*XToken*-- TYPE = {none, whiteSpace, nonToken};
SleepOrDestroy: --*StarWindowShell*-- PROCEDURE [Handle] RETURNS [Handle];
Slide: --*Window*-- PROCEDURE [window: Handle, newPlace: Place];

SlideAndSize: --*Window*-- PROCEDURE [
  window: Handle, newBox: Box, gravity: Gravity ^nw];
SlideAndSizeAndStack: --*Window*-- PROCEDURE [
  window: Handle, newBox: Box, newSibling: Handle, newParent: Handle ^NIL,
  gravity: Gravity ^nw];
SlideAndStack: --*Window*-- PROCEDURE [
  window: Handle, newPlace: Place, newSibling: Handle, newParent: Handle ^NIL];
SmallPictureProc: --*Containee*-- TYPE = PROCEDURE [
  data: DataHandle ^NIL, type: NSFile.Type ^ignoreType,
  normalOrReference: PictureState] RETURNS [smallPicture: XString.Character];
SocketNumber: --*XFormat*-- PROCEDURE [
  h: Handle ^NIL, socketNumber: System.SocketNumber, format: NetFormat];
SortOrder: --*XString*-- TYPE = MACHINE DEPENDENT{
  standard, spanish, swedish, danish, firstFree, null(255)};
SourceModifyProc: --*ContainerWindow*-- TYPE = PROCEDURE [
  window: Window.Handle, source: ContainerSource.Handle]
  RETURNS [changeInfo: ContainerSource.ChangeInfo];
spares: --*BWSAttributeTypes*-- CARDINAL = 20;
SpecialIndex: --*XLReal*-- TYPE = NATURAL;
Spinnaker: --*ProductFactoringProducts*-- Product = 2;
SqRt: --*XLReal*-- PROCEDURE [Number] RETURNS [Number];
Stack: --*Window*-- PROCEDURE [
  window: Handle, newSibling: Handle, newParent: Handle ^NIL];
StandardClose: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [Handle];
StandardCloseAll: --*StarWindowShell*-- PROCEDURE [sws: Handle] RETURNS [Handle];
StandardCloseEverything: --*StarWindowShell*-- PROCEDURE
  RETURNS [notClosed: Handle];
StandardFilterState: --*XToken*-- TYPE = ARRAY [0..1] OF UNSPECIFIED;
StandardLimitProc: --*StarWindowShell*-- LimitProc;
Star: --*ProductFactoringProducts*-- Product = 0;
State: --*StarWindowShell*-- TYPE = MACHINE DEPENDENT{
  awake, sleeping, dead, last(7)};
StatusOfFill: --*ContainerCache*-- PROCEDURE [cache: Handle]
  RETURNS [CacheFillStatus];
StopOrNot: --*XString*-- TYPE = {stop, not} ^not;
Store: --*Cursor*-- PROCEDURE [h: Handle];
StoreCharacter: --*Cursor*-- PROCEDURE [c: XChar.Character];
StoreNumber: --*Cursor*-- PROCEDURE [n: CARDINAL];
StoreTable: --*TIPStar*-- PROCEDURE [Placeholder, TIP.Table] RETURNS [TIP.Table];
StreakNature: --*XChar*-- TYPE = {leftToRight, rightToLeft};
StreakSuccession: --*FormWindow*-- TYPE = SimpleTextDisplay.StreakSuccession;
StreakSuccession: --*SimpleTextDisplay*-- TYPE = {
  leftToRight, rightToLeft, fromFirstChar};
StreamObject: --*XFormat*-- PROCEDURE [sH: Stream.Handle] RETURNS [Object];
StreamProc: --*XFormat*-- FormatProc;
StreamToHandle: --*XToken*-- PROCEDURE [s: Stream.Handle] RETURNS [h: Handle];
String: --*XFormat*-- PROCEDURE [h: Handle ^NIL, s: LONG STRING];
StringArray: --*XMessage*-- TYPE = LONG DESCRIPTOR FOR ARRAY CARDINAL OF
  XString.ReaderBody;
StringIntoBuffer: --*SimpleTextDisplay*-- PROCEDURE [
  string: XString.Reader, bufferProc: BufferProc, lineWidth: CARDINAL ^177777B,
  wordBreak: BOOLEAN ^TRUE, streakSuccession: StreakSuccession ^fromFirstChar,
  font: SimpleTextFont.MappedFontHandle ^NIL]
  RETURNS [lastLineWidth: CARDINAL, result: Result, rest: XString.ReaderBody];
StringIntoWindow: --*SimpleTextDisplay*-- PROCEDURE [
  string: XString.Reader, window: Window.Handle, place: Window.Place,

```
   lineWidth: CARDINAL ^177777B, maxNumberOfLines: CARDINAL ^1,
   lineToLineDeltaY: CARDINAL ^0, wordBreak: BOOLEAN ^TRUE,
   flags: BitBlt.BitBltFlags ^LOOPHOLE[42000B]]
   RETURNS [lines: CARDINAL, lastLineWidth: CARDINAL];
StringOfItem: --ContainerSource-- StringOfItemProc;
StringOfItemProc: --ContainerSource-- TYPE = PROCEDURE [
   source: Handle, itemIndex: ItemIndex, stringIndex: CARDINAL]
   RETURNS [XString.ReaderBody];
StuffCharacter: --TIP-- PROCEDURE [
   window: Window.Handle, char: XString.Character] RETURNS [BOOLEAN];
StuffCurrentSelection: --TIP-- PROCEDURE [window: Window.Handle]
   RETURNS [BOOLEAN];
StuffResults: --TIP-- PROCEDURE [window: Window.Handle, results: Results];
StuffSTRING: --TIP-- PROCEDURE [window: Window.Handle, string: LONG STRING]
   RETURNS [BOOLEAN];
StuffString: --TIP-- PROCEDURE [window: Window.Handle, string: XString.Reader]
   RETURNS [BOOLEAN];
StuffTrashBin: --TIP-- PROCEDURE [window: Window.Handle] RETURNS [BOOLEAN];
Subtract: --XLReal-- PROCEDURE [a: Number, b: Number] RETURNS [Number];
SubtractItem: --MenuData-- PROCEDURE [menu: MenuHandle, old: ItemHandle];
SubtractPopupMenu: --StarWindowShell-- PROCEDURE [
   sws: Handle, menu: MenuData.MenuHandle];
Subtype: --Prototype-- TYPE = CARDINAL;
Swap: --BlackKeys-- PROCEDURE [old: Keyboard, new: Keyboard];
Swap: --Cursor-- PROCEDURE [old: Handle, new: Handle];
Swap: --SoftKeys-- PROCEDURE [
   window: Window.Handle, table: TIP.Table ^NIL,
   notifyProc: TIP.NotifyProc ^NIL, labels: Labels ^xxx,
   highlightedKey: CARDINAL ^nullKey, outlinedKey: CARDINAL ^nullKey];
Swap: --StarWindowShell-- PROCEDURE [
   new: Handle, old: Handle, poppedProc: PoppedProc ^NIL];
SwapExistingFormWindows: --PropertySheet-- PROCEDURE [
   shell: StarWindowShell.Handle, new: Window.Handle, apply: BOOLEAN ^TRUE,
   newMenuItemProc: MenuItemProc ^NIL, newMenuItems: MenuItems ^LOOPHOLE[0],
   newTitle: XString.Reader ^NIL, newAfterTakenDownProc: MenuItemProc ^NIL]
   RETURNS [old: Window.Handle];
SwapFormWindows: --PropertySheet-- PROCEDURE [
   shell: StarWindowShell.Handle, newFormWindowItems: FormWindow.MakeItemsProc,
   newFormWindowItemsLayout: FormWindow.LayoutProc ^NIL, apply: BOOLEAN ^TRUE,
   destroyOld: BOOLEAN ^TRUE, newMenuItemProc: MenuItemProc ^NIL,
   newMenuItems: MenuItems ^LOOPHOLE[0], newTitle: XString.Reader ^NIL,
   newGlobalChangeProc: FormWindow.GlobalChangeProc ^NIL,
   newAfterTakenDownProc: MenuItemProc ^NIL] RETURNS [old: Window.Handle];
SwapItem: --MenuData-- PROCEDURE [
   menu: MenuHandle, old: ItemHandle, new: ItemHandle];
SwapItemProc: --MenuData-- TYPE = PROCEDURE [
   menu: MenuHandle, old: ItemHandle, new: ItemHandle];
SwapMenuItem: --Attention-- PROCEDURE [
   old: MenuData.ItemHandle, new: MenuData.ItemHandle];
Switches: --XToken-- FilterProcType;
SyntaxError: --XToken-- SIGNAL [r: XString.Reader];
systemFileCatalog: --BWSFileTypes-- NSFile.Type = 10476B;
systemFontHeight: --SimpleTextDisplay-- READONLY CARDINAL;
```

```
Table: --TIP-- TYPE = LONG POINTER TO TableObject;
TableError: --TIP-- TYPE = {fileNotFound, badSyntax};
TableObject: --TIP-- TYPE;
TabStops: --FormWindow-- TYPE = RECORD [
   variant: SELECT type: TabType FROM
      fixed => [interval: CARDINAL],
      vary => [list: LONG DESCRIPTOR FOR ARRAY CARDINAL OF CARDINAL],
      ENDCASE];
TabType: --FormWindow-- TYPE = {fixed, vary};
Take: --ContainerSource-- TakeProc;
TakeNEXTKey: --FormWindow-- PROCEDURE [window: Window.Handle, item: ItemKey];
TakeProc: --ContainerSource-- TYPE = PROCEDURE [
   source: Handle, copyOrMove: Selection.CopyOrMove,
   afterHint: ItemIndex ^nullItem, withinSameSource: BOOLEAN ^FALSE,
   changeProc: ChangeProc ^NIL, changeProcData: LONG POINTER ^NIL,
   selection: Selection.ConvertProc ^NIL] RETURNS [ok: BOOLEAN];
Tan: --XLReal-- PROCEDURE [radians: Number] RETURNS [tan: Number];
Target: --Selection-- TYPE = MACHINE DEPENDENT{
   window, shell, subwindow, string, length, position, integer, interpressMaster,
   file, fileType, token, help, keyboard, interscriptScript, interscriptFragment,
   serializedFile, name, firstFree, last(1023)};
textFlags: --Display-- BitBltFlags;
TextHintAction: --FormWindow-- TYPE = {replace, append, nil};
TextHintsProc: --FormWindow-- TYPE = PROCEDURE [
   window: Window.Handle, item: ItemKey]
   RETURNS [
      hints: LONG DESCRIPTOR FOR ARRAY CARDINAL OF XString.ReaderBody,
      freeHints: FreeTextHintsProc, hintAction: TextHintAction ^replace];
ThumbFlavor: --StarWindowShell-- TYPE = {downClick, track, upClick};
ThumbScrollProc: --StarWindowShell-- TYPE = PROCEDURE [
   sws: Handle, vertical: BOOLEAN, flavor: ThumbFlavor, m: INTEGER,
   outOfN: INTEGER];
Ticket: --Containee-- TYPE [2];
timeOnly: --XTime-- XString.Reader;
TIPResults: --SimpleTextEdit-- PROCEDURE [f: Field, results: TIP.Results]
   RETURNS [tookInputFocus: BOOLEAN, changed: BOOLEAN];
TotalOrPartial: --ContainerSource-- TYPE = {total, partial};
Trajectory: --Display-- PROCEDURE [
   window: Handle, box: Window.Box ^xxx, proc: TrajectoryProc,
   source: LONG POINTER ^NIL, bpl: CARDINAL ^16, height: CARDINAL ^16,
   flags: BitBltFlags ^bitFlags, missesChildren: BOOLEAN ^FALSE,
   brick: Brick ^xxx];
TrajectoryProc: --Display-- TYPE = PROCEDURE [Handle]
   RETURNS [Window.Box, INTEGER];
TransitionProc: --StarWindowShell-- TYPE = PROCEDURE [
   sws: Handle, state: State];
Trapezoid: --Display-- TYPE = RECORD [
   x: Interpolator, y: INTEGER, w: Interpolator, h: NATURAL];
TreatNumbersAs: --XTime-- TYPE = {
   dayMonthYear, monthDayYear, yearMonthDay, yearDayMonth, dayYearMonth,
   monthYearDay};
TrimBoxStickouts: --Window-- PROCEDURE [window: Handle, box: Box] RETURNS [Box];
TTYObject: --XFormat-- PROCEDURE [h: TTY.Handle] RETURNS [Object];
TTYProc: --XFormat-- FormatProc;
```

```
Type: --Context-- TYPE = MACHINE DEPENDENT{
   all, first, lastAllocated(37737B), last(37777B)};
Type: --Cursor-- TYPE = MACHINE DEPENDENT{
   blank, bullseye, confirm, ftpBoxes, hourGlass, lib, menu, mouseRed, pointDown,
   pointLeft, pointRight, pointUp, questionMark, scrollDown, scrollLeft,
   scrollLeftRight, scrollRight, scrollUp, scrollUpDown, textPointer,
   groundedText, move, copy, sameAs, adjust, row, column, last(255)};
UnderChangedProc: --Window-- TYPE = PROCEDURE [Handle, Box];
Unintelligible: --XTime-- ERROR [vicinity: CARDINAL];
UniqueAction: --Selection-- PROCEDURE RETURNS [Action];
UniqueTarget: --Selection-- PROCEDURE RETURNS [Target];
UniqueType: --Context-- PROCEDURE RETURNS [type: Type];
UniqueType: --Cursor-- PROCEDURE RETURNS [Type];
Units: --UnitConversion-- TYPE = MACHINE DEPENDENT{
   inch, mm, cm, mica, point, pixel, pica, didotPoint, cicero,
   seventySecondOfAnInch, last(15)};
unknownContext: --XString-- Context;
UnmapFont: --SimpleTextFontExtra-- PROCEDURE [SimpleTextFont.MappedFontHandle];
Unpack: --XTime-- PROCEDURE [
   time: System.GreenwichMeanTime ^defaultTime, ltp: LTP ^useSystem]
   RETURNS [unpacked: Unpacked];
Unpacked: --XTime-- TYPE = RECORD [
   year: [0..4070B],
   month: [0..11],
   day: [0..31],
   hour: [0..23],
   minute: [0..59],
   second: [0..59],
   weekday: [0..6],
   dst: BOOLEAN,
   zone: System.LocalTimeParameters];
UnpostedSwapItemProc: --MenuData-- SwapItemProc;
UnsignedDecimalFormat: --XFormat-- NumberFormat;
UnterminatedQuote: --XToken-- SIGNAL;
Update: --ContainerWindow-- PROCEDURE [window: Window.Handle];
UpperCase: --XChar-- PROCEDURE [c: Character] RETURNS [Character];
useGMT: --XTime-- useThese LTP;
UserAbort: --TIP-- PROCEDURE [Window.Handle] RETURNS [BOOLEAN];
userPassword: --StarDesktop-- Atom.ATOM;
useSystem: --XTime-- useSystem LTP;
Valid: --Window-- PROCEDURE [Handle] RETURNS [BOOLEAN];
Validate: --Window-- PROCEDURE [window: Handle];
ValidateReader: --XString-- PROCEDURE [r: Reader];
ValidateTree: --Window-- PROCEDURE [window: Handle ^rootWindow];
ValidExponent: --XLReal-- TYPE = [-512..511];
Value: --Selection-- TYPE = RECORD [
   value: LONG POINTER,
   ops: LONG POINTER TO ValueProcs ^NIL,
   context: LONG UNSPECIFIED ^0];
ValueCopyMoveProc: --Selection-- TYPE = PROCEDURE [
   v: ValueHandle, op: CopyOrMove, data: LONG POINTER];
ValueFreeProc: --Selection-- TYPE = PROCEDURE [v: ValueHandle];
ValueHandle: --Selection-- TYPE = LONG POINTER TO Value;
ValueProcs: --Selection-- TYPE = RECORD [
   free: ValueFreeProc ^NIL, copyMove: ValueCopyMoveProc ^NIL];
```

VanillaArrowScroll: --*StarWindowShell*-- ArrowScrollProc;
vanillaContext: --*XString*-- Context;
vanillaScrollData: --*StarWindowShell*-- ScrollData;
VanillaThumbScroll: --*StarWindowShell*-- ThumbScrollProc;
version: --*BWSAttributeTypes*-- NSFile.ExtendedAttributeType = 10460B;
Version: --*Prototype*-- TYPE = CARDINAL;
VersionColumn: --*FileContainerSourceExtra*-- PROCEDURE
   RETURNS [attribute FileContainerSource.ColumnContentsInfo];
ViewPoint: --*ProductFactoringProducts*-- Product = 5;
ViewPointApps: --*ProductFactoringProducts*-- Product = 6;
Visibility: --*FormWindow*-- TYPE = {visible, invisible, invisibleGhost};
WaitSeconds: --*TIP*-- PROCEDURE [seconds: CARDINAL];
When: --*StarWindowShell*-- TYPE = {before, after};
White: --*Display*-- PROCEDURE [
   window: Handle, box: Window.Box, bounds: Window.BoxHandle ^NIL];
WhiteSpace: --*XToken*-- FilterProcType;
WordsForBitmapUnder: --*Window*-- PROCEDURE [window: Handle] RETURNS [CARDINAL];
Writer: --*XString*-- TYPE = LONG POINTER TO WriterBody;
WriterBody: --*XString*-- TYPE = PRIVATE MACHINE DEPENDENT RECORD [
   context(0:0..15): Context,
   limit(1:0..15): CARDINAL,
   offset(2:0..15): CARDINAL,
   bytes(3:0..31): Bytes,
   maxLimit(5:0..15): CARDINAL,
   endContext(6:0..15): Context,
   zone(7:0..31): UNCOUNTED ZONE];
WriterBodyFromBlock: --*XString*-- PROCEDURE [
   block: Environment.Block, inUse: CARDINAL ^0] RETURNS [WriterBody];
WriterBodyFromNSString: --*XString*-- PROCEDURE [
   s: NSString.String, homogeneous: BOOLEAN ^FALSE] RETURNS [WriterBody];
WriterBodyFromSTRING: --*XString*-- PROCEDURE [
   s: LONG STRING, homogeneous: BOOLEAN ^FALSE] RETURNS [WriterBody];
WriterInfo: --*XString*-- PROCEDURE [w: Writer]
   RETURNS [unused: CARDINAL, endContext: Context, zone: UNCOUNTED ZONE];
WriterObject: --*XFormat*-- PROCEDURE [w: XString.Writer] RETURNS [Object];
WriterProc: --*XFormat*-- FormatProc;
XFormatObject: --*MessageWindow*-- PROCEDURE [window: Window.Handle]
   RETURNS [o: XFormat.Object];
xorBoxFlags: --*Display*-- BitBltFlags;
xorFlags: --*Display*-- BitBltFlags;
xorGrayFlags: --*Display*-- BitBltFlags;
zero: --*XLReal*-- Number;
Zone: --*Undo*-- PROCEDURE RETURNS [UNCOUNTED ZONE];

name is the name of the table.

fillinByRow determines what happens when the user presses the NEXT key. If **fillInByRow** is TRUE, pressing the NEXT key advances through the table one row at a time, and the table is expanded by rows. In this case, the number of columns is fixed and the number of rows can be either fixed or varying. If **fillinByRow** is FALSE, then pressing the NEXT key advances through the table one column at a time, and the table is expanded by columns. In this case, the number of rows is fixed and the number of columns can be either fixed or varying. **fixedRows** and **fixedColumns** indicate whether the user can change the number of rows and columns in the table.

numberOfColumns and **numberOfRows** are used as hints for **StartTable**.

visibleHeader indicates whether there should be a visible header at the top of the table; **repeatHeader**, **repeatTopCaption**, **repeatBottomCaption** indicates whether or not to repeat these items on every page if the table occupies multiple pages.

borderLine describes the table border (not the frame border), and **dividerLine** describes the line between the header row and the rest of the table. A line can have a width anywhere from one pixel to six pixels.

```
Line: TYPE = RECORD [
    linestyle: Linestyle,
    linewidth: Linewidth];

Linestyle: TYPE = MACHINE DEPENDENT{
    zero(0), solid, dashed, dotted, double, broken, firstAvailable, lastAvailable(255)};
```